

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Integración de dispositivos BLE con el Asistente Virtual Alexa mediante tecnologías IoT

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención: Sistemas Electrónicos
Autor: D. Javier Araña Melián
Tutores: D. Valentín De Armas Sosa
D. Félix B. Tobajas Guerrero
Fecha: Enero 2019



UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Integración de dispositivos BLE con el Asistente Virtual Alexa mediante tecnologías IoT

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: _____

Vocal

Secretario/a

Fdo.: _____

Fdo.: _____

Fecha: Enero 2019

Índice de Contenido

Acrónimos	15
1. Introducción	1
1.1.1 Antecedentes de IoT.....	1
1.2 Antecedentes BLE.....	4
1.3 Antecedentes de los Asistentes Virtuales	6
1.4 Objetivos	7
1.5 Estructura de la memoria.....	8
2. Photon	11
2.1 Descripción y características	11
2.2 Diagrama de bloques y patillaje	12
2.3 Conectando el dispositivo Photon.....	15
2.4 Modos del dispositivo	17
3. Herramientas software	21
3.1 Particle Web IDE.....	21
3.2 Particle Cloud	24
4. Alexa.....	27
4.1 Skill Interface	27
4.2 Skill Service	29
4.3 Test	35
5. Integración del dispositivo Photon con el software de Alexa	39
5.1 Configuración de la Skill Interface y Skill Service de Alexa.....	39
5.2 Configuración de los servicios de Amazon Alexa	44
5.3 Software Photon.....	47
6. Bluetooth Low Energy	51
6.1 Introducción	51
6.2 Topología de red.....	52
6.3 Arquitectura	53
6.3.1 Capa Física	55
6.3.2 Capa de enlace	55
6.3.3 Interfaz de Control de Host	57
6.3.4 Protocolo de Control y Adaptación de Enlace Lógico	57
6.3.5 Protocolo de Atributos	57
6.3.6 Administrador de seguridad.....	58
6.3.7 Perfil de atributo genérico	58
6.3.8 Perfil de Acceso Genérico.....	64

6.4	Dispositivo BLE	65
6.5	Alexa Echo Dot	66
6.6	Diagrama de flujo	67
7.	RedBear DUO	69
7.1	Descripción y características	69
7.2	Diagrama de bloques y patillaje	70
7.3	Conectando el DUO	72
7.4	Reclamar el DUO	78
8.	Integración del PLAYBULB Candle con Alexa mediante el dispositivo RedBear DUO.....	81
8.1	Descubrimiento de los valores del servidor GATT	81
8.2	<i>Alexa Skill Interface</i>	88
8.3	<i>Alexa Skill Service</i>	89
8.4	Firmware del dispositivo DUO.....	91
8.5	Validación.....	112
9.	Conclusiones.....	125
	Referencias.....	127
	Pliego de Condiciones	131
	Condiciones Hardware	131
	Condiciones Software.....	131
	Condiciones firmware	132
	Presupuesto	133
	Trabajo tarifado por tiempo empleado	133
	Amortización del inmovilizado material	134
	Amortización del material hardware	134
	Amortización del material software.....	135
	Redacción del trabajo.....	136
	Aplicación de impuestos y coste total.....	136

Índice de Figuras

Figura 1. Paradigma del IoT como el resultado de las convergencias de diferentes puntos de vista [2]	3
Figura 2. Esquema de conexión de dispositivos	7
Figura 3. Particle Photon	11
Figura 4. Patillaje del Photon.....	12
Figura 5. Diagrama del patillaje del Photon I	15
Figura 6. Diagrama del patillaje del Photon II	15
Figura 7. Diagrama del patillaje del Photon III	15
Figura 8. Photon conectado a fuente de alimentación	16
Figura 9. Particle App - Tinker	17
Figura 10. Inicio de sesión/registro en Particle Build	21
Figura 11. Particle Build (Web IDE)	22
Figura 12. Particle Apps.....	23
Figura 13. Particle Devices	24
Figura 14. Barra de estado	24
Figura 15. Alexa Skills	27
Figura 16. AlexaSkill.js (I).....	30
Figura 17. AlexaSkill.js (II).....	30
Figura 18. AlexaSkill.js (III).....	32
Figura 19. AlexaSkill.js (IV).....	33
Figura 20. AlexaSkill.js (V).....	34
Figura 21. AlexaSkill.js (VI).....	34
Figura 22. AlexaSkill.js (VII).....	35
Figura 23. Probando Test (I)	36
Figura 24. Probando Test (II)	37
Figura 25. Configuración del hardware del Photon	39
Figura 26. Intent Schema Photon.....	39
Figura 27. Index.js Photon (I).....	41
Figura 28. Index.js Photon (II).....	41
Figura 29. Index.js Photon (III).....	42
Figura 30. Index.js Photon (IV)	43
Figura 31. package.json Photon	43
Figura 32. Alexa Developer Console.....	45
Figura 33. AWS Lambda.....	45
Figura 34. Resultado test state intent	46
Figura 35. Test select intent	46
Figura 36. Resultado test select intent.....	46
Figura 37. Código software Photon.....	48
Figura 38. Echosim.io	49
Figura 39. Modo de operación Broadcast.....	52
Figura 40. Modo de operación Connections.....	53
Figura 41. Pila de protocolos del estándar BLE.....	54
Figura 42. Canales de frecuencia.....	55
Figura 43. Máquina de estados de la capa de enlace.....	56
Figura 44. Jerarquía de datos y atributos.....	61
Figura 45. Declaración del servicio.....	62

Figura 46. Declaración y valor de una característica.....	62
Figura 47. Valor de la declaración de características.....	63
Figura 48. Propiedades de una característica.....	63
Figura 49. PLAYBULB Candle.....	65
Figura 50. Alexa Echo Dot.....	66
Figura 51. Diagrama de flujo Alexa, DUO y PLAYBULB Candle.....	67
Figura 52. Diagrama de flujo BLE Central.....	68
Figura 53. Diagrama de bloques del DUO.....	70
Figura 54. Patillaje del DUO.....	70
Figura 55. Iniciando RedBear Duo App (con internet y sin internet).....	72
Figura 56. Configuración del DUO con las funciones WiFi y BLE activadas/desactivadas.....	73
Figura 57. Instrucciones para configurar el DUO a través de WiFi.....	73
Figura 58. Configuración del DUO a través de BLE.....	74
Figura 59. Configuración del DUO a través de una red WiFi cercana.....	74
Figura 60. Identificación del DUO y versión del firmware actual.....	75
Figura 61. Conectando el DUO a una red WiFi.....	76
Figura 62. Conectando a una red WiFi un DUO enlazado por BLE.....	76
Figura 63. Aplicación ejecutándose en el DUO.....	77
Figura 64. Dispositivos conectados a Particle Cloud.....	78
Figura 65. Reclamo del dispositivo.....	79
Figura 66. Renombrando el dispositivo.....	79
Figura 67. nRFConnect Scan.....	81
Figura 68. nRFConnect Servicios.....	82
Figura 69. nRFConnect Características.....	83
Figura 70. Dispositivo CC2540 USB Dongle.....	83
Figura 71. Aplicación PLAYBULBX.....	84
Figura 72. Paquetes BLE estableciendo conexión.....	84
Figura 73. Paquetes BLE con trama de apagado del LED del PLAYBULB Candle.....	85
Figura 74. Paquetes BLE con trama de cambio de color del LED a rojo.....	85
Figura 75. Paquetes BLE con trama de cambio de color del LED a verde.....	86
Figura 76. Paquetes BLE con trama de cambio de color del LED a azul.....	86
Figura 77. Identificación de características con el nRFConnect.....	87
Figura 78. Identificando la característica de notificaciones con nRFConnect.....	88
Figura 79. Intent Schema de la solución final.....	88
Figura 80. Index.js solución final (I).....	89
Figura 81. Index.js solución final (II).....	90
Figura 82. Index.js solución final (III).....	90
Figura 83. Firmware del dispositivo DUO (I).....	91
Figura 84. Firmware del dispositivo DUO (II).....	92
Figura 85. Firmware del dispositivo DUO (III).....	93
Figura 86. Firmware del dispositivo DUO (IV).....	94
Figura 87. Estructura de datos advertisementReport_t.....	94
Figura 88. Firmware del dispositivo DUO (V).....	96
Figura 89. Firmware del dispositivo DUO (VI).....	97
Figura 90. Firmware del dispositivo DUO (VII).....	98
Figura 91. Firmware del dispositivo DUO (VIII).....	99
Figura 92. Estructura de datos service.....	100
Figura 93. Firmware del dispositivo DUO (IX).....	101

Figura 94. Estructura de datos characteristic.....	102
Figura 95. Firmware del dispositivo DUO (X)	103
Figura 96. Estructura de datos descriptor	104
Figura 97. Firmware del dispositivo DUO (XI)	105
Figura 98. Firmware del dispositivo DUO (XII)	106
Figura 99. Firmware del dispositivo DUO (XIII)	107
Figura 100. Firmware del dispositivo DUO (XIV)	108
Figura 101. Firmware del dispositivo DUO (XV)	108
Figura 102. Firmware del dispositivo DUO (XVI)	109
Figura 103. Firmware del dispositivo DUO (XVII)	110
Figura 104. Firmware del dispositivo DUO (XVIII)	110
Figura 105. Firmware del dispositivo DUO (XIX)	111
Figura 106. Firmware del dispositivo DUO (XX)	112
Figura 107. Establecimiento de conexión entre dispositivos	112
Figura 108. Identificación de los servicios del dispositivo Peripheral	113
Figura 109. Identificación de las características del servicio primario (I)	114
Figura 110. Identificación de las características del servicio primario (II)	114
Figura 111. Identificación de las características del servicio secundario	115
Figura 112. Identificación de los descriptores de los servicios del dispositivo Peripheral	115
Figura 113. Apagado del LED	116
Figura 114. Encendido del LED	116
Figura 115. Cambio de color del LED.....	117
Figura 116. Tramas de apagado del LED	118
Figura 117. Tramas de encendido del LED	118
Figura 118. Tramas de cambio de color del LED a rojo	118
Figura 119. Tramas de cambio de color del LED a verde	119
Figura 120. Tramas de cambio de color del LED a azul	119
Figura 121. Tramas de solicitud y respuesta de lectura de una característica	119
Figura 122. Test de la función setLightColor	120
Figura 123. Test de la función setLightState	121
Figura 124. Resultado del test de la función setLightState con el slot down	121
Figura 125. Resultado del test de la función setLightState con el slot on	122
Figura 126. Resultado del test de la función setLightColor con el slot red	122
Figura 127. Resultado del test de la función setLightColor con el slot green	122
Figura 128. Resultado del test de la función setLightColor con el slot blue	123

Índice de Tablas

Tabla 1. Descripción patillaje Photon.....	14
Tabla 2. Comparativa BLE y Bluetooth convencional.....	51
Tabla 3. Condiciones Hardware.....	131
Tabla 4. Condiciones Software	131
Tabla 5. Condiciones Firmware	132
Tabla 6. Factor de corrección	134
Tabla 7. Amortización del material hardware.....	135
Tabla 8. Amortización del material software	135
Tabla 9. Coste total inmovilizado material.....	135
Tabla 10. Presupuesto.....	136
Tabla 11. Presupuesto total del Trabajo Fin de Grado.....	136

Acrónimos

ADC	<i>Analog-to-Digital Converter</i>
ARPANET	<i>Advanced Research Projects Agency Network</i>
ATT	<i>Attribute protocol</i>
BLE	<i>Bluetooth Low Energy</i>
COITT	<i>Colegio Oficial de Ingenieros de Telecomunicación)</i>
DAC	<i>Digital-to-Analog Converter</i>
EITE	<i>Escuela de Ingeniería de Telecomunicación y Electrónica</i>
FHSS	<i>Frequency Hopping Spread Spectrum</i>
GAP	<i>Generic Acces Profile</i>
GATT	<i>Generic Attribute Profile</i>
GPIO	<i>General Purpose Input/Output</i>
HCI	<i>Host Controller Interface</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IoE	<i>Internet of Everything</i>
IoT	<i>Internet of Things</i>
ITU	<i>International Telecommunication Union</i>
L2CAP	<i>Logical Link Control and Adaptation Protocol</i>
LED	<i>Light Emitting Diode</i>
LL	<i>Link Layer</i>
PAN	<i>Personal Area Network</i>
PCB	<i>Printed Circuit Board</i>
PHY	<i>Physical Layer</i>
PWM	<i>Pulse-Width Modulation</i>
RAM	<i>Random Access Memory</i>
RF	<i>Radio Frequency</i>
RTC	<i>Real Time Clock</i>
SIG	<i>Special Interest Group</i>
SMP	<i>Security Manager Protocol</i>
SMPS	<i>Switch Mode Power Supply</i>
SRAM	<i>Static Random Access Memory</i>

TCP/IP	<i>Transmission Control Protocol/Internet Protocol</i>
TFG	Trabajo Fin de Grado
ULPGC	Universidad de Las Palmas de Gran Canaria
UUID	<i>Universally Unique Identifier</i>
VDC	<i>Voltage Direct Current</i>
WICED	<i>Wireless Internet Connectivity for Embedded System</i>
WPAN	<i>Wireless Personal Area Network</i>

MEMORIA

1. Introducción

Esta sección se divide en varios apartados, en los cuales se analizan y discuten diversos aspectos relativos a la justificación y el planteamiento del presente TFG (Trabajo Fin de Grado).

1.1.1 Antecedentes de IoT

Actualmente vivimos en un mundo de continuo cambio a nivel tecnológico, basta con encender el ordenador o el móvil para poder comprobar que cada vez son más las cosas que es posible hacer con ellos, y es que hoy en día se puede enviar a hacer la colada a una lavandería, reservar un billete de avión para un vuelo, o pagar la cuenta del restaurante con solo pulsar con un dedo. Incluso sentados en el sillón de casa, es posible pedir a la televisión que busque una película en concreto y la reproduzca solo con decirlo en voz alta. La tecnología avanza con el ser humano y para el ser humano, intentando que la vida sea más sencilla y cómoda teniendo interconectados todos los dispositivos del día a día, y permitiendo que puedan ser más productivos. Y para poder tener estos dispositivos conectados entre sí hace falta un medio: Internet. Internet hace que unas cosas se comuniquen con otras. Que transfieran datos de su estado y reciban otros del entorno, creando así una red de información que puede ser usada de diferentes formas y aprovechada para hacer funcionar todo en su conjunto, componiendo finalmente el Internet de las Cosas (*Internet of things* o IoT).

Según *Cisco Internet Business Solutions Group*, el Internet de las Cosas surgió entre 2008 y 2009 para definir ese momento en el que habían más cosas conectadas a Internet que personas. De acuerdo con la ITU (*International Telecommunication Union*) [1], para poder llevar a cabo un repaso rápido por los acontecimientos que han marcado el nacimiento y evolución de IoT, hay que remontarse a 1926 para hablar de Nikola Tesla, cuyas patentes y trabajos teóricos aportaron la base de las comunicaciones inalámbricas y de radio. En 1969 se envió el primer mensaje a través de ARPANET (*Advanced Research Projects Agency Network*), una red operativa que con el paso del tiempo dio origen a la Internet global de la actualidad. Diez años después se probó el protocolo TCP/IP (*Transmission Control Protocol/Internet Protocol*), en el que se basa la Internet actual y que permiten la transmisión de datos entre ordenadores. En 1990 Berners-Lee implementó la primera comunicación exitosa entre un cliente HTTP (*Hypertext Transfer Protocol*) y un servidor a través de Internet, había inventado la WWW (*World Wide Web*). Él mismo, un año más tarde, creó la primera página web, hecho que dio lugar a un desarrollo tecnológico vertiginoso, comenzando la revolución de Internet.

En 1999 Kevin Ashton, impartió una conferencia en *Procter & Gamble* donde habló por primera vez del concepto de Internet de las Cosas. En los primeros años de este siglo el término es

mencionado en publicaciones de vanguardia como *The Guardian*, *Scientific American* y *The Boston Globe*. También se despliega la tecnología RFID de forma masiva, por parte del Departamento de Defensa de los EEUU, y los almacenes Walmart a nivel comercial, ésta tecnología tiene un papel importantísimo en el mundo de IoT, ya que es fundamental en la perspectiva orientada a cosas, la cual se explicará más adelante. En 2005 la agencia de la ITU (*International Telecommunications Union*) publica el primer estudio sobre el tema. A partir de ese momento Internet de las Cosas adquiere otro nivel. En 2005 también comienza la aventura de Arduino, clave para desarrollar elementos autónomos, o bien conectarse a otros dispositivos o interactuar con otros programas, tanto con el hardware como con el software.

En 2006 se comercializa el Nabaztag (liebre, en armenio), un pequeño conejo que se conecta a Internet por ondas Wifi, originalmente fabricado por la empresa francesa *Violet*. Este conejo es capaz de comunicarse con su usuario emitiendo mensajes de voz, luminosos o moviendo sus orejas, difundiendo informaciones como la meteorología, la Bolsa, la calidad del aire, la llegada de los correos electrónicos, etc... En 2008 un grupo de empresas se unen para crear la *IPSO Alliance* con el objetivo de promover el uso del protocolo de Internet en redes de objetos inteligentes y hacer posible IoT. Actualmente en IPSO participan 59 empresas de todo el mundo como Bosch, Cisco, Ericsson, Motorola, Google, Toshiba o Fujitsu. En 2008 comienza el proyecto Pachube (en 2011 fue adquirido por *LogMeIn*, líder en la provisión de soluciones de nube). En 2010 el primer ministro chino Wen Jiabao dijo que IoT era la clave de la industria para China. En 2011 se lanzó el nuevo protocolo IPV6. Samsung, Google, Nokia y otros fabricantes anuncian sus proyectos NFC. Se crea la iniciativa *IoT-GSI Global Standards* para promover la adopción de estándares para IoT a escala global. China continua invirtiendo e impulsando el desarrollo y la investigación en Internet de las Cosas con instituciones como *Shanghai Institute* o la *Chinese Academy of Sciences*.

El término IoT hace referencia a varios aspectos sobre la extensión de la red y el mundo de los objetos físicos, lo cual es posible gracias a la aparición generalizada de sensores y actuadores [2]. El concepto de IoT, introducido a finales de la década de los noventa por Kevin Ashton, es multidisciplinar y se ve afectado por distintos puntos de vista [3]. Acorde a esta idea, IoT puede ser definido mediante la intersección de tres perspectivas diferentes: orientada a objetos, orientada a Internet, y orientada a semántica, tal y como se muestra en la Figura 1.

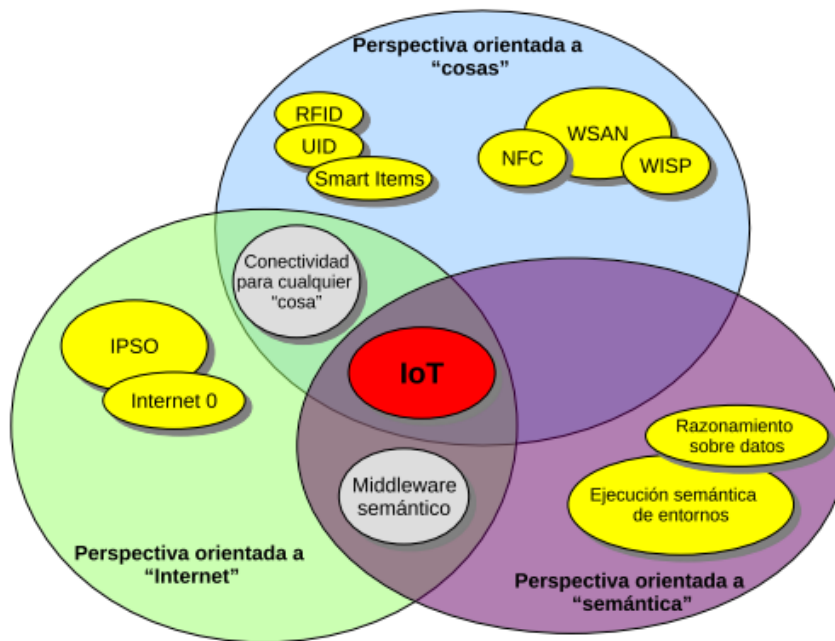


Figura 1. Paradigma del IoT como el resultado de las convergencias de diferentes puntos de vista [2]

Estos objetos comparten una serie de características, entre las que se pueden destacar: [4]

- Forman parte del conjunto de objetos físicos reales.
- Son capaces de recibir y emitir mensajes, también poseen mecanismos que permiten que otros dispositivos los localicen y contacten.
- Poseen un identificador único.
- Tienen mecanismos de computación, lo que les permite procesar información de diversa índole en función de la capacidad del objeto.
- Poseen medios para detectar magnitudes físicas. Algunos podrán ser configurados para comportarse también como actuadores que desencadenen una acción en el mundo real.

La visión orientada a Internet comprende los conceptos de la red IoT, caracterizándola como un conjunto muy grande de nodos en un sistema distribuido. Los nodos recopilan información del mundo físico y la traducen para su envío. Teniendo en cuenta el volumen de información, será importante proveer a la red de medios que garanticen su escalabilidad y estabilidad.

La intersección entre la visión orientada a Internet y la visión orientada a objetos trata de unir de alguna manera los objetos físicos con las topologías de red para lograr que exista comunicación entre todas las cosas. Para que esto sea posible es necesario que la red sea capaz de asignar direcciones únicas a cada uno de los objetos, además de aportar a los objetos cotidianos mecanismos de conectividad.

Al considerar la cantidad de objetos cotidianos a los que se podría proporcionar conectividad para incluirlos en el paradigma IoT, se prevé como algo imprescindible el disponer de un grupo muy grande y heterogéneo de nodos enviando, recibiendo y almacenando información. Por tanto, los mecanismos propios de la visión semántica para buscar, almacenar, conectar y representar toda la información generada por la red también son necesarios. Dentro de la rama semántica se tratan los temas de estandarización y definición de lenguajes para la comunicación en IoT.

Hace más de diez años que se habla del concepto de IoT. La tecnología IoT trae consigo cambios significativos en múltiples sectores, desde la sanidad hasta el mercado de consumo, pasando por el ámbito del hogar. IoT es un concepto que representa a un conjunto de dispositivos de múltiples ámbitos, los cuales generan enormes flujos de datos que se comparten entre sí, permitiendo la posibilidad de actuar en consecuencia y culminando en una nueva generación de dispositivos inteligentes [5]. Este concepto, que inicialmente parecía ciencia ficción, está marcando, cada vez más, su lugar en la sociedad actual, hasta tal punto, que las grandes empresas líderes en este tipo de tecnologías, como *International Business Machines* (IBM) o Cisco [6,7], han desarrollado sus propias soluciones para campos como el de la industria, por ejemplo, para el análisis de cantidades ingentes de datos (*big data*), reconocimiento de patrones, y aprendizaje automático. No es de extrañar que el concepto de IoT comience a tomar presencia en nuestra vida cotidiana, donde evoluciona para convertirse en *Internet of Everything* (IoE).

1.2 Antecedentes BLE

Junto con estos nuevos conceptos surgen nuevas necesidades, como la de conectar los dispositivos a redes inalámbricas que no penalicen mucho la autonomía de los terminales. Es por ello que surgen las WPAN (*Wireless Personal Area Network*) para sustituir las necesidades de interconectar dispositivos de modo que puedan compartir datos y voz de forma sencilla, eficiente y de bajo coste sin grandes consumos de energía [8]. Las redes PAN (*Personal Area Network*) fue tratado por primera vez por los ingenieros de IBM, siendo pioneros en este tema al plantear un modelo de transmisión de datos utilizando como medio de transmisión la conductividad natural del cuerpo humano. Este hecho supuso un problema, ya que siempre debía haber contacto entre la persona y el dispositivo. De este problema, surge la necesidad de usar enlaces inalámbricos como la radiofrecuencia o los infrarrojos, lo que propició el nacimiento de las redes WPAN, cuyo objetivo principal siempre fue brindar a los dispositivos que se encuentren cerca geográficamente, estén o no en movimiento, la facultad para intercambiar información sin necesidad de cables. Una de las peculiaridades de este tipo de redes, es que pueden coexistir con

otras tecnologías inalámbricas, proporcionándole al usuario la posibilidad de conectarse utilizando la mejor opción disponible.

Las WPAN ofrecen la posibilidad de interconectar dispositivos de manera sencilla, sin impactar en el consumo de energía de forma rápida y segura. Por ello, se ha incrementado la presencia en dispositivos móviles de tecnologías como *Bluetooth*. *Bluetooth* es un estándar de comunicación que usa señales de radio de corto alcance con el objetivo de eliminar los cables a la hora de interconectar dispositivos y se inició en 1994 por L. M. Ericsson. Este estándar fue nombrado *Bluetooth* en honor al rey de la mitología nórdica que lleva el mismo nombre. El *Bluetooth SIG (Special Interest Group)* fue fundado por Ericsson, IBM, Intel, Nokia y Toshiba en febrero de 1998. A día de hoy, hay más de 1900 compañías que pertenecen al SIG, entre las que destacan 3Com, Motorola, Lucent, etc.

En la actualidad, se han publicado estándares tales como *Bluetooth Low Energy (BLE)*, Zigbee o NFC que cumplen perfectamente con estas condiciones, en función de la distancia. *Bluetooth Low Energy*, también denominado *Bluetooth Smart* o *Bluetooth 4.1*, es un nuevo sistema inalámbrico de interconexión de dispositivos de bajo consumo de potencia. Esta nueva tecnología no solo ha mejorado el consumo, sino que también ha mejorado el radio de transferencia, pudiendo alcanzar hasta los 100 metros. Del mismo modo, también ha mejorado la velocidad de transferencia que ahora llega hasta 1Mbps [9,10]. La versión 4 de *Bluetooth*, o BLE, fue desarrollada por el grupo de investigación de Nokia cuando estos se dieron cuenta de que el estándar *Bluetooth* actual no era capaz de cubrir ciertas áreas de aplicación, lo que les llevó a desarrollar una nueva tecnología basada en *Bluetooth*, ofreciendo un coste y un consumo de energía menor, dando como resultado el estándar *Bluetooth Low End Extension*. Nokia se unió con otras compañías y en octubre de 2006 lanzaron otra tecnología llamada Wibre. Un año más tarde, en junio de 2007, tras varias negociaciones con los miembros del *Bluetooth SIG* se llegó a un acuerdo para incluir Wibre en una futura versión de *Bluetooth* de bajo consumo, la cual se hizo oficial en diciembre de 2009 dando acogida a la próxima versión que sería conocida como BLE.

BLE se implementó bajo la idea de que los nuevos dispositivos que portasen BLE fueran totalmente compatibles con el resto de los dispositivos que utilizaban los estándares de *Bluetooth* anteriores, de forma que no quedara estancado el desarrollo de la tecnología *Bluetooth* [11]. El funcionamiento de BLE es muy parecido al *Bluetooth* convencional en lo que se refiere a los procedimientos de anuncio y sincronización, diferenciándose del *Bluetooth* convencional en la inclusión de protocolos que aseguran la comunicación entre dispositivos [12]. BLE utiliza la misma técnica del salto de frecuencia de espectro ensanchado y opera bajo la misma banda de

frecuencias ISM que el *Bluetooth* clásico (2.4 a 2.48 GHz), con una pequeña diferencia en la distribución de los canales: el *Bluetooth* clásico utiliza 79 canales de 1MHz de ancho de banda, mientras que en BLE se utilizan 40 de 2 MHz de ancho de banda [13].

1.3 Antecedentes de los Asistentes Virtuales

Los servicios de asistencia son anteriores a la Asistencia Virtual. Aunque estos últimos son una evolución de los primeros, es importante comprender que los conceptos son diferentes. La diferencia fundamental entre la Asistencia Virtual y los servicios de asistencia radica en la plataforma sobre la cual se brindan los servicios. Es importante destacar que los servicios de asistencia están dentro del negocio de los trabajos orientados a tareas específicas, en los que no necesariamente se conoce bien al cliente o el negocio de los clientes y que, por lo tanto, colaboran en los proyectos sin demasiado aporte. Por el contrario, un Asistente Virtual se involucra personalmente en una relación continua con sus clientes que trasciende la realización de un trabajo en particular.

El predecesor de Siri, Cortana, Alexa o *Google Assistant* fue Audrey. El sistema de reconocimiento automático del habla, desarrollado por los Laboratorios Bell en 1952. Este sistema era capaz de reconocer números absolutos.

Diez años más tarde, la compañía IBM desarrolló Shoebox, que procesaba los números del 0 al 9 y las operaciones matemáticas “más”, “menos”, “subtotal”, “total”, “falso” y “de” para dar respuestas a peticiones como “Cinco más tres más ocho más seis más cuatro menos nueve, total”, y ofrecer la respuesta correcta “17”. El siguiente hito llegó de la mano de la Universidad de Carnegie Mellon en 1978 bajo el nombre de Harpy, y tenía capacidad para entender 1.000 palabras. En la década de los 80, las investigaciones se centraron en el desarrollo de sistemas basados en el modelo *Hidden Markov Model*, es decir, en la probabilidad estadística de que determinada palabra siga a otra, con el objetivo de dejar atrás los sistemas basados en palabras para crear aplicaciones capaces de entender frases.

En la última década del siglo XX, compañías como IBM, Dragon, Philips, Lernout & Hauspie o Microsoft, llevaron el reconocimiento de voz al ordenador. La evolución tecnológica sufrió un importante parón con el estallido de la crisis de las puntocom y no fue hasta 2011 cuando se presentó el primer asistente virtual digital avanzando incorporado al iPhone 4S, Siri. Nacido de un proyecto de investigación financiado por DARPA, el sistema de Apple utiliza inteligencia artificial, tecnologías de búsqueda de datos geográficos, y herramientas de lenguaje, entre otras.

Por otro lado, asociado con las tecnologías IoT, también surgen necesidades como la de controlar los dispositivos utilizando la voz, desde simplemente sustituir acciones del usuario para agilizar determinadas tareas, tales como “llama a esta persona” o “avisame de esto”, hasta tareas más creativas, como, por ejemplo: “coge las fotos del viaje y móntalas con música de fondo” [14]. Con el objetivo de cubrir esta necesidad, muchas de las grandes compañías de este sector han lanzado al mercado asistentes virtuales inteligentes, por ejemplo: Facebook con FacebookM [15], Apple con Siri [16], Google con GoogleNow [17], Microsoft con Cortana [18], o Amazon con Alexa; Así, Amazon lanzó recientemente al mercado Alexa [19], un asistente personal virtual que cada día se hace más inteligente. Alexa está incluida en los dispositivos Amazon Echo y Echo Dot, entre otros, y ofrece al usuario la posibilidad de controlar productos, escuchar música y más. Sin embargo, a pesar de que el hardware de estos dispositivos soporta *Bluetooth* hasta ahora no está habilitada para interactuar con dispositivos con BLE [20], lo cual supone un problema a la hora de interconectar múltiples dispositivos, pues la tecnología BLE está cada día más presente en los aparatos electrónicos de corto alcance, debido a su bajo consumo de potencia.

1.4 Peticionario

Actúa como peticionario del presente Trabajo Fin de Grado la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC) como requisito indispensable para la obtención del título de Grado en Ingeniería en Tecnologías de la Telecomunicación, tras haber superado con éxito las asignaturas especificadas en el Plan de Estudios.

1.5 Objetivos

El objetivo principal de este TFG consiste en lograr la integración de Alexa, el asistente personal virtual de Amazon disponible en un dispositivo Amazon Echo Dot, con dispositivos BLE. Puesto que anteriormente se ha comentado que la conexión directa de estos dispositivos no es un hecho compatible, se utilizará una plataforma de interconexión de dispositivos IoT que hará de pasarela entre ambos, tal y como se muestra en la Figura 2.

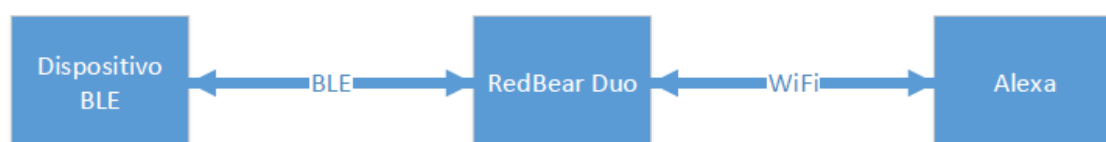


Figura 2. Esquema de conexión de dispositivos

La plataforma de interconexión de dispositivos IoT que se va a utilizar es *RedBear DUO*. Este dispositivo combina un potente microcontrolador ARM Cortex M3 con un chip Wi-Fi Broadcom y otro chip *Bluetooth 4.1 Dual Mode*. Además, posee otras características destacables como son su alimentación a 3,3 VDC, sus 18 entradas/salidas de propósito general (GPIO), plataforma IoT en la nube, y que su diseño es open source, lo que facilita su integración en un producto. Por otro lado, la herramienta software que se va a utilizar para la implementación del TFG es Particle IDE de Particle. Esta herramienta utiliza la URL como un localizador uniforme de recursos (Uniform Source Locator), donde el recurso es el dispositivo en cuestión. Cada dispositivo tiene una URL, la cual se puede utilizar para obtener variables, realizar llamadas de una función o actualizar el *firmware*.

El objetivo principal de este TFG es conseguir realizar la integración del dispositivo *PLAYBULB Candle* con el asistente virtual Alexa, dada la incompatibilidad inicial de Alexa para comunicarse con dispositivos que gestionan sus comunicaciones mediante el protocolo BLE, utilizando el dispositivo *RedBear DUO* como pasarela entre ambos para tratar de reproducir mediante comandos de voz, algunas de las funcionalidades que ofrece la aplicación móvil del dispositivo *PLAYBULB Candle*.

Además, se utilizará en la fase inicial de desarrollo del presente TFG otra plataforma de interconexión de dispositivos denominada *Photon*, la cual ofrece una solución de desarrollo de IoT completa con un potente microcontrolador ARM Cortex M3 con un chip WiFi WICED (*Wireless Internet Connectivity for Embedded Devices*) de Cypress. El microcontrolador y el chip de Wi-Fi están alojados en un pequeño módulo en miniatura llamado PØ (P-cero). El dispositivo *Photon* se basa en la arquitectura WICED de Cypress y está listo para funcionar con una fuente de alimentación SMPS de 3.3VDC. El dispositivo *Photon* también tiene componentes de RF e interfaz al PØ en una pequeña PCB de una cara.

1.6 Estructura de la memoria

En el capítulo 1 de esta memoria se detallarán los antecedentes de las tecnologías implicadas en el desarrollo del TFG, del mismo modo, también se especifican los objetivos propuestos. El capítulo 2, da una visión general de las características y el funcionamiento del dispositivo *Photon*, mientras que en el capítulo 3, se especifica el funcionamiento de las herramientas software utilizadas durante el desarrollo del TFG.

En el capítulo 4 del presente documento, se definen los aspectos más importantes de Alexa, entre los cuales se encuentran la definición de las *skills* y la elaboración de los test que corroboran su correcto funcionamiento. El capítulo 5, muestra detalladamente el procedimiento

utilizado para elaborar la integración del dispositivo *Photon* con el software de Alexa. El capítulo 6, describe el funcionamiento de la topología y la arquitectura de la tecnología BLE.

El capítulo 7 del presente documento, aporta un enfoque general de las características y el funcionamiento del dispositivo *RedBear DUO*, mientras que el capítulo 8, detalla la estructura hardware del proyecto y se explica en profundidad el software de la solución final, incluyendo la configuración utilizada para integrar la tecnología BLE y Alexa. En el capítulo 9, se establecen las conclusiones obtenidas tras la realización del TFG, así como las posibles líneas futuras de desarrollo del presente TFG.

A continuación, se presentará la bibliografía utilizada en este TFG. Los apartados finales, se encargarán de exponer las condiciones bajo las que se ha desarrollado el presente TFG, así como los gastos generados en la realización del trabajo.

2. Dispositivo Photon

Durante el desarrollo del proyecto se trabajó con dos plataformas de interconexión de dispositivos: *Photon* de Particle y *DUO* de RedBear. A pesar de que en el modelo final del TFG solo se utiliza el *DUO*, el dispositivo *Photon* se utilizó durante la primera parte del TFG para llevar a cabo un desarrollo inicial que sentó las bases de conocimiento de la plataforma de interconexión de dispositivos y desgranó el funcionamiento de los servicios que ofrecen las *skills* de Alexa. Por tanto, a lo largo de este apartado se detallarán las partes fundamentales del dispositivo *Photon*.

2.1 Descripción y características

El dispositivo *Photon* es una plataforma de desarrollo que combina un potente microcontrolador ARM Cortex M3 con un chip Broadcom Wi-Fi en un pequeño módulo en miniatura llamado PØ (*P-zero*). Particle agrega una sólida fuente de alimentación SMPS (*Switch Mode Power Supply*) de 3.3VDC (*Voltage Direct Current*), RF (*Radio Frequency*) y componentes de interfaz de usuario al PØ en una pequeña placa PCB (*Printed Circuit Board*) de una sola cara llamado dispositivo *Photon*. El diseño es de código abierto, por lo que el Photon estará siempre listo para integrarse con cualquier producto [21]. La Figura 3 muestra una imagen del dispositivo Photon.

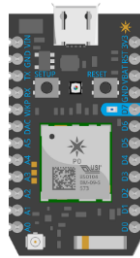


Figura 3. Particle Photon

A continuación, se describen las principales características del dispositivo *Photon*:

- Particle PØ utiliza el módulo Wi-Fi Broadcom BCM43362 y trabaja con los estándares Wi-Fi 802.11b / g / n.
- Posee el chip STM32F205RGY que trabaja en la frecuencia de 120Mhz y utiliza el microcontrolador ARM Cortex M3.
- Incorpora 1MB de memoria *flash* y 128KB de RAM (*Random Access Memory*).
- Incorpora 1 LED (*Light Emitting Diode*) RGB de estado.
- Dispone de 18 pines GPIO (*General Purpose Input/Output*) de señal mixta y periféricos avanzados.
- Diseño de código abierto.
- Sistema operativo en tiempo real.

- Tiene los certificados FCC, CC e IC.

2.2 Diagrama de bloques y patillaje

En la Figura 4 se muestra el patillaje del dispositivo *Photon*, cuyas descripciones se incluyen en la Tabla 1.

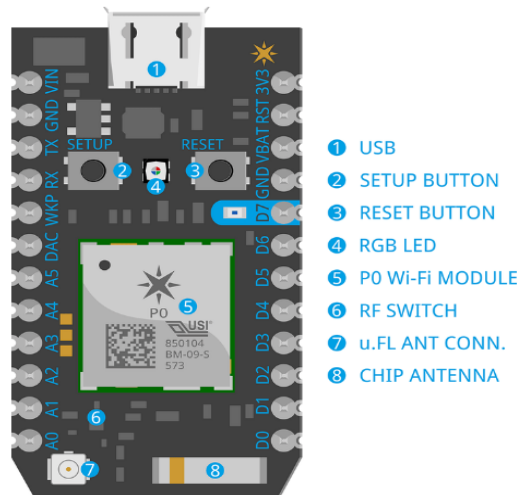


Figura 4. Patillaje del Photon.

PIN	Descripción
VIN	Este pin se puede usar tanto de entrada como de salida. Como entrada, se debe suministrar de 3.6 a 5.5VDC para alimentar el <i>Photon</i> . Cuando el <i>Photon</i> se alimenta a través del puerto USB, este pin emitirá un voltaje de aproximadamente 4.8VDC debido a un diodo Schottky de protección de polaridad inversa entre <i>VUSB</i> y <i>VIN</i> . Cuando se usa como salida, la carga máxima en <i>VIN</i> es 1A.
RST	Entrada de reinicio, activa a nivel bajo. La circuitería de la placa presenta una resistencia de 1k ohm entre <i>RST</i> y <i>3V3</i> , y un condensador de 0,1 uF entre <i>RST</i> y <i>GND</i> .
VBAT	Suministro al RTC (<i>Real Time Clock</i>) interno, registros de respaldo y SRAM (<i>Static Random</i>

	Acces Memory) cuando 3V3 no está presente (1.65 a 3.6VDC).
3V3	<p>Este pin es la salida del regulador de la placa y está conectado internamente a la alimentación del módulo de Wi-Fi. Al alimentar el <i>Photon</i> a través de <i>VIN</i> o el puerto USB, este pin emitirá un voltaje de 3.3VDC. Este pin también se puede utilizar para alimentar el <i>Photon</i> directamente (entrada máxima 3.3VDC). Cuando se usa como salida, la carga máxima en 3V3 es 100 mA.</p> <p>NOTA: Al encender el <i>Photon</i> a través de este pin, asegúrese de que la alimentación esté desconectada de <i>VIN</i> y <i>USB</i> o provocará daños en la placa.</p>
RX	Principalmente utilizado como UART RX, pero también se puede usar como un GPIO digital o PWM (<i>Pulse-width modulation</i>)
TX	Principalmente utilizado como UART TX, pero también se puede usar como un GPIO digital o PWM.
WKP	WAKEUP que a nivel alto activa el módulo cuando se encuentra en modo de suspensión/espera. Cuando no se usa como WAKEUP, este pin también se puede usar como una entrada digital GPIO, ADC (<i>Analog-to-Digital Converter</i>) o PWM. Se puede denominar <i>A7</i> cuando se usa como un ADC.
DAC	Salida digital a analógica (D/A) de 2 bits (0-4095), denominada <i>DAC</i> (<i>Digital-to-Analog Converter</i>) o <i>DAC1</i> en el software. También

	se puede usar como GPIO digital o ADC. Se puede denominar A6 cuando se usa como un ADC. A3 es una segunda salida DAC utilizada como DAC2 en el software.
A0~A7	Entradas analógicas a digitales (A/D) de 12 bits (0-4095), y también GPIO digitales. A6 y A7 son asignaciones de conveniencia del código, lo que significa que los pines realmente no están etiquetados como tales, pero se pueden usar funciones como analogRead (A7) para determinar su valor. A6 se mapea en el pin DAC y A7 en el pin WKP. A4, A5, A7 también pueden usarse como salida PWM.
D0~D7	Solo pines GPIO digitales. D0 ~ D3 también se puede usar como salida PWM.

Tabla 1. Descripción patillaje Photon

Además de los 24 pines que hay alrededor de la parte exterior del dispositivo *Photon*, hay 7 pines en la parte inferior de la placa del *Photon* que se pueden usar para conectar señales adicionales como: salidas del LED RGB, un botón SETUP, una línea de activación SMPS y USB D + / D-. Los pines *Photon* # 25-31 se describen en los diagramas de patillaje. PWM está disponible en los pines *D0*, *D1*, *D2*, *D3*, *A4*, *A5*, *WKP*, *RX*, *TX*, pero tienen una pequeña condición a tener en cuenta: el temporizador PWM se duplica en dos pines (*A5* / *D2*) y (*A4* / *D3*) para un total de 7 salidas PWM independientes. Por ejemplo: PWM puede usarse en *A5* mientras que *D2* se usa como GPIO, o *D2* como PWM, mientras que *A5* se utiliza como entrada analógica. Sin embargo, *A5* y *D2* no se pueden usar como salidas PWM controladas independientemente al mismo tiempo [22]. A continuación, la Figura 5, la Figura 6 y la Figura 7 muestran los diagramas del patillaje del dispositivo *Photon*.

Pin	USB	Exposed Functions			STM32 Pin	PØ Pin #	PØ Pin Name	
VIN		VIN						
GND		GND						
TX	P H O T O N		USART1_TX	TIM1_CH2	PA9	39	MICRO_UART_TX	
RX			USART1_RX	TIM1_CH3	PA10	38	MICRO_UART_RX	
WKP				TIM5_CH1	PA0	27	MICRO_WKUP	
DAC		ADC0			DAC1	PA4	22	MICRO_SPI_SSN ^[2]
A5		ADC4				PA7	23	MICRO_SPI_MOSI
A4		ADC6	SPI (MOSI)		TIM3_CH2	PA6	25	MICRO_SPI_MISO
A3		ADC7	SPI (MISO)		TIM3_CH1			
A2		ADC5	SPI (SCK)			PA5	24	MICRO_SPI_SCK
A1		ADC12	SPI (SS)			PC2	2	MICRO_GPIO_6
A0		ADC13				PC3	1	MICRO_GPIO_7
		ADC15			PC5	54	MICRO_GPIO_8	

Figura 5. Diagrama del patillaje del Photon I

USB	Pin	Exposed Functions			STM32 Pin	PØ Pin #	PØ Pin Name	
	3V3	3V3						
	RST	RST			E8	26	MICRO_RST_N	
	VBAT	VBAT			A9	28	VBAT	
	GND	GND						
P H O T O N	D7	JTAG_TMS			PA13	44	MICRO_JTAG_TMS	
	D6	JTAG_TCK			PA14	40	MICRO_JTAG_TCK	
	D5	JTAG_TDI	SPI1 (SS)		I2S3_WS	PA15	43	MICRO_JTAG_TDI
	D4	JTAG_TDO	SPI1 (SCK)		I2S3_SCK	PB3	41	MICRO_JTAG_TDO
	D3	JTAG_TRST	SPI1 (MISO)		TIM3_CH1	PB4	42	MICRO_JTAG_TRSTN
	D2		SPI1 (MOSI)	CAN2_RX	TIM3_CH2	PB5	3	MICRO_GPIO_5
	D1	SCL		CAN2_TX	TIM4_CH1	PB6	5	MICRO_GPIO_3
	D0	SDA			TIM4_CH2	PB7	4	MICRO_GPIO_4

Figura 6. Diagrama del patillaje del Photon II

	User I/O	Photon Pin #	Exposed Functions		STM32 Pin	PØ Pin #	PØ Pin Name	
P H O T O N	RGB LED - RED	27		TIM2_CH2	PA1	8	MICRO_GPIO_0	
	RGB LED - GREEN	28		TIM2_CH3	PA2	7	MICRO_GPIO_1	
	RGB LED - BLUE	29		TIM2_CH4	PA3	6	MICRO_GPIO_2	
	Setup Button	26		TIM3_CH2	I2S3_MCK	PC7	53	MICRO_GPIO_9
	Reset Button	23			E8	26	MICRO_RST_N	
	USB Data+	31			PB15	51	MICRO_USB_HS_DP	
	USB Data-	30			PB14	52	MICRO_USB_HS_DM	
	SMPS Enable	25						
	Peripheral Key		ADC	SPI	PWM/Servo/Tone			
			JTAG	SPI1	I2S	DAC		
		I2C/Wire	Serial1	CAN				

Figura 7. Diagrama del patillaje del Photon III

2.3 Conectando el dispositivo Photon

Ahora que se ha alcanzado un cierto conocimiento sobre el dispositivo *Photon*, el siguiente paso es conectarlo a Internet. Para ello se debe conectar el cable USB a su fuente de alimentación, en este caso basta con conectarlo al ordenador, tal y como se muestra en la Figura 8.



Figura 8. Photon conectado a fuente de alimentación

Tan pronto como esté enchufado, el LED RGB del dispositivo *Photon* debería comenzar a parpadear en azul. En el caso de que el *Photon* no esté parpadearo en azul, se debe mantener presionado el botón *SETUP*. Si, por el contrario, el *Photon* no está parpadearo en absoluto, o si el LED está parpadearo un color naranja apagado, es posible que no esté recibiendo suficiente energía, una posible solución es intentar cambiar la fuente de alimentación o cable USB [23].

Una vez llegados a este punto, existen diferentes maneras de seguir. En esta memoria se describirán los pasos que se han utilizado a lo largo del desarrollo del TFG. Por tanto, el siguiente paso consiste en conectar el *Photon* a Internet utilizando un *smartphone*, para lo cual hay que descargar *iOS/Android Particle App* en el *smartphone*. Abrir la aplicación, iniciar sesión o registrarse para obtener una cuenta con Particle si no se tiene una. A continuación, presiona el ícono “+” y selecciona el dispositivo que se desea agregar. Luego, sigue las instrucciones en la pantalla para conectar el dispositivo a la red Wi-Fi. Este proceso puede demorarse un poco.

La primera vez que el dispositivo *Photon* se conecta, parpadeará violeta durante unos minutos mientras descarga las actualizaciones, incluso pueden pasar de 6 a 12 minutos hasta que se completen las actualizaciones, dependiendo de la conexión a Internet, este proceso puede que lleve al *Photon* a reiniciarse algunas veces. No se debe reiniciar ni desenchufar el *Photon* durante este proceso, ya que se si lo hace, es posible que deba reparar el dispositivo.

Una vez que se haya conectado el dispositivo, éste habrá guardado esa red. El dispositivo *Photon* puede almacenar hasta cinco redes, para agregar una nueva red después de haber realizado la configuración inicial, debe poner el *Photon* en *modo de escucha* nuevamente y proceder tal y como se indicó anteriormente (omitiendo la parte del reclamo si así lo desea). En el caso de que considere que el dispositivo tiene demasiadas redes, puede borrarle la memoria de cualquier red Wi-Fi que haya aprendido. Para ello, mantenga presionado el botón *SETUP* durante 10 segundos hasta que el LED RGB parpadee en azul rápidamente, indicando que se han

eliminado todos los perfiles. La aplicación Particle ahora debería estar en la pantalla Tinker, como se muestra en la Figura 9.



Figura 9. Particle App - Tinker

Como puede ver en la Figura 9, los círculos representan diferentes pines del dispositivo Photon. Al tocar estos círculos, puede ver las funciones de Tinker disponibles para los pines asociados. Con el objetivo de demostrar que se ha realizado el reclamo correctamente, se puede usar Tinker y la aplicación de Particle para interactuar con cualquier pin del dispositivo. El pin *D7* ya viene conectado a un pequeño LED azul en la parte frontal del *Photon*, al configurar la potencia del pin *D7* en alto, este LED se enciende. Para ello hay que tocar *D7* y luego *digitalWrite* en la ventana emergente, ahora cuando se toca el círculo *D7* el pequeño LED azul debería apagarse o encenderse.

2.4 Modos del dispositivo

En este apartado se echará un vistazo a los comportamientos de los diferentes modos que pueden presentar los dispositivos que se han utilizado durante el desarrollo del TFG. Los modos juegan un papel fundamental, dado que son capaces de indicar el estado actual del dispositivo. Teniendo en cuenta que existen muchos modos del dispositivo, solo se incluirán en la memoria los modos estándar. Estos modos representan los comportamientos más típicos de los dispositivos por medio de diferentes patrones de la luz del LED que contiene [24].

- Conectado

Mientras esté parpadeando cian, el dispositivo estará conectado a Internet. Cuando está en este modo, puede llamar a funciones y código de *flash*.

- Actualización de *firmware* OTA

Si el dispositivo está parpadeando en magenta, estará cargando una aplicación o actualizando su *firmware*. Este estado se desencadena por una actualización de firmware o por un código parpadeante del Web IDE o Desktop IDE. Tenga en cuenta que, si ingresa a este modo manteniendo presionado *SETUP* al inicio, el parpadeo magenta indica que al soltar el botón *SETUP* se ingresará al modo seguro para conectarse a la nube y no ejecutar el *firmware* de la aplicación.

- Buscando Internet

Si está parpadeando en verde, está intentando conectarse a Internet.

- Conectando a la nube

Cuando el dispositivo está en proceso de conectarse a la nube, parpadeará rápidamente en cian.

- Modo de escucha

En este modo, está esperando la entrada para conectarse a WiFi. El dispositivo Photon necesita estar en modo de escucha para comenzar a conectarse con la aplicación móvil o por USB. Para activar el modo de escucha, es necesario mantener presionado el botón *SETUP* durante tres segundos, hasta que el LED RGB comience a parpadear en azul.

- Modo seguro

El modo seguro conecta el dispositivo a la nube, pero no ejecuta ningún *firmware* de la aplicación. Este modo es uno de los más útiles para el desarrollo o para la solución de problemas. Si algo sale mal con la aplicación que se cargó en el dispositivo, es posible configurarlo en modo a prueba de fallos. Esto ejecuta el SO del dispositivo, pero no ejecuta ningún código de aplicación, lo que puede ser útil si el código de la aplicación contiene errores que impiden que el dispositivo se conecte a la nube. El dispositivo indica que está en modo seguro con el LED parpadeando magenta.

Para ponerlo en modo seguro se debe:

1. Mantener presionado AMBOS botones.
2. Soltar solo el botón *RESET*, mientras mantienes presionado el botón *SETUP*.
3. Esperar a que el LED comience a parpadear magenta.
4. Soltar el botón *SETUP*.

Antes de ingresar al modo seguro, el dispositivo tratará de conectarse a la nube y si no puede conectarse a la nube, no podrá entrar en modo seguro. El dispositivo automáticamente entrará en

modo seguro si no se muestra un código de aplicación en el dispositivo o si la aplicación no es válida.

- Modo DFU (actualización de *firmware* del dispositivo)

Para programar el dispositivo con un *firmware* personalizado a través de USB, deberá usar este modo. Este modo activa el gestor de arranque incorporado que acepta archivos binarios de *firmware* a través de *dfu-util*.

Para ingresar al modo DFU debe:

1. Mantener presionado AMBOS botones.
2. Soltar solo el botón *RESET*, mientras mantienes presionado el botón *SETUP*.
3. Esperar a que el LED comience a parpadear en amarillo (primero parpadeará en magenta).
4. Soltar el botón *SETUP*.

El dispositivo ahora está en el modo DFU.

3. Herramientas software

3.1 Particle Web IDE

Particle Web IDE es una herramienta que permite reprogramar los dispositivos [25]. Para poder acceder a ella es necesario iniciar sesión o registrarse en caso de no disponer de una cuenta.

Crear una cuenta es un proceso simple de un solo paso. Cuando se presente la pantalla de inicio de sesión, hay que hacer clic en "*create one now*" y completar el formulario, tal y como se muestra en la Figura 10.

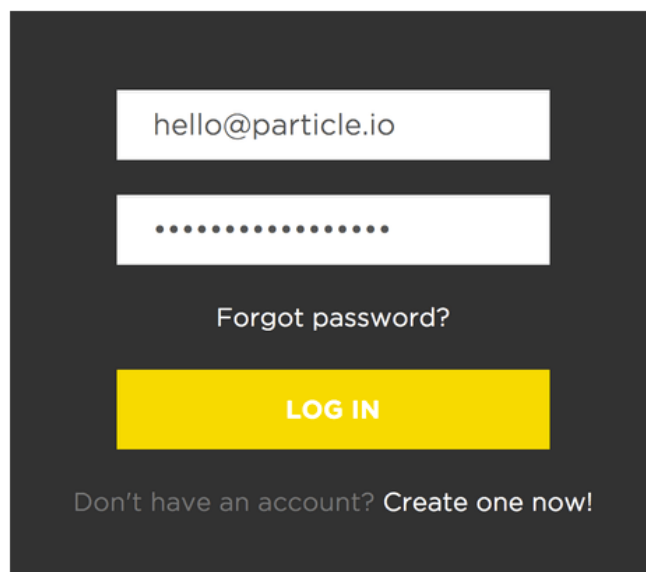
The image shows a login/registration form on a dark background. It features two white input fields: the top one contains the email address 'hello@particle.io' and the bottom one contains a series of dots representing a password. Below the password field is the text 'Forgot password?'. A prominent yellow button with the text 'LOG IN' is centered below the form. At the bottom of the form area, there is a link that says 'Don't have an account? Create one now!'.

Figura 10. Inicio de sesión/registro en Particle Build

Particle Build es un entorno de desarrollo integrado o IDE, lo cual implica desarrollar software en una aplicación fácil de usar, que se ejecuta en el navegador web. Particle Build comienza mostrando la barra de navegación de la izquierda de la Figura 11.

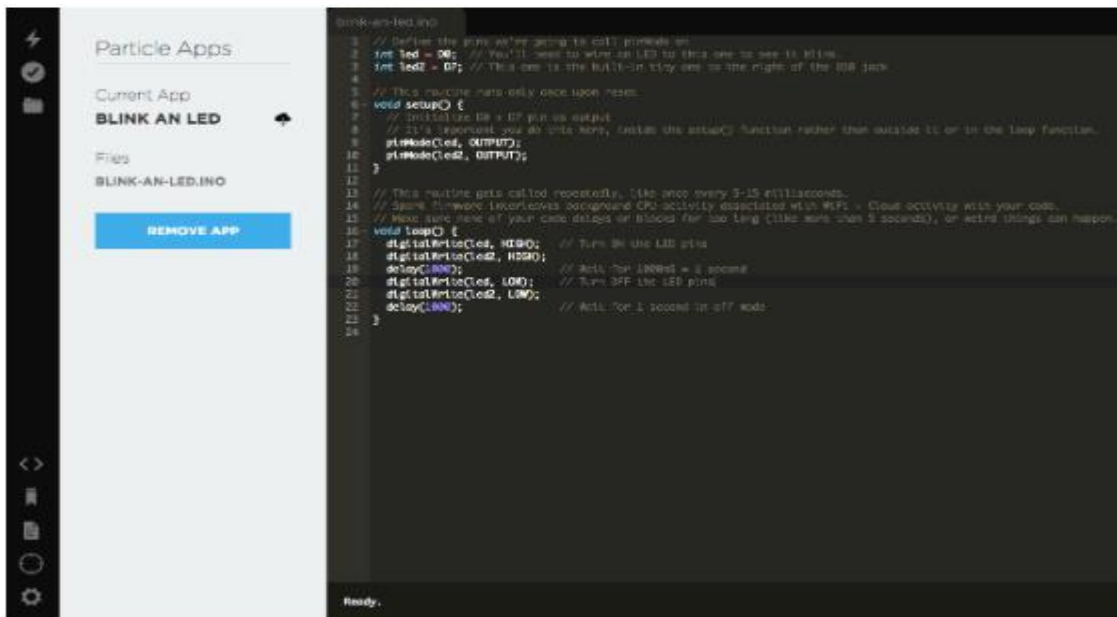


Figura 11. Particle Build (Web IDE)

En la parte superior, hay tres botones que cumplen funciones importantes:

- *Flash*: muestra el código actual en el dispositivo. Esto inicia una actualización de *firmware* inalámbrica y carga el nuevo software.
- *Verify*: este botón compila el código sin actualizarlo al dispositivo; si hay algún error de código, se mostrará en la consola de depuración en la parte inferior de la pantalla.
- *Save*: guarda cualquier cambio que haya realizado en el código.

En la parte inferior, hay cuatro botones más para navegar a través del IDE:

- *Code*: muestra una lista de las aplicaciones de *firmware* y permite seleccionar cuál editar / *flashear*.
- *Library*: habilita la exploración de las bibliotecas enviadas por otros usuarios y permite desarrollar bibliotecas propias.
- *Docs*: conduce a la documentación de Particle.
- *Devices*: muestra una lista de los dispositivos, en la que se puede elegir cuál *flashear*/usar y obtener más información de cada dispositivo.
- *Settings*: Permite cambiar la contraseña, cerrar la sesión u obtener el token de acceso para llamadas API.

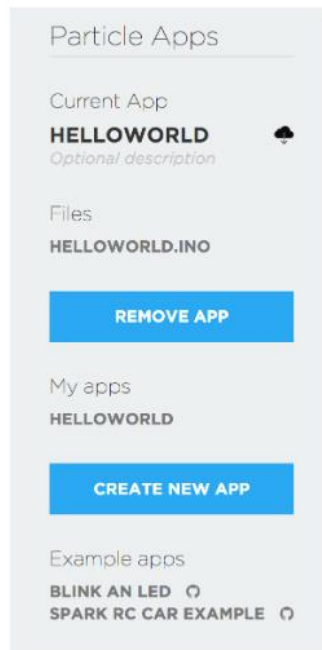


Figura 12. Particle Apps

En la Figura 12 se incorpora la sección *Particle Apps*, capaz de mostrar el nombre de la aplicación actual que está usando en el editor, así como una lista de las demás aplicaciones, entre las que se incluyen aplicaciones de ejemplo compatibles con la comunidad. La aplicación abierta en el editor se muestra debajo del encabezado *Current App*. Desde este panel, se puede acceder a muchos botones y acciones disponibles que facilitan el trabajo y la administración de las bibliotecas de aplicaciones.

A continuación, se van a explicar los pasos a seguir para escribir código en el Web IDE.

1. Conectar: Encender el dispositivo y asegurarse de que esté parpadeando cian.
2. Obtener código: El código puede ser tanto propio, como de los ejemplos de la comunidad.
3. Seleccionar dispositivo: en caso de tener más de un dispositivo, hay que asegurarse de que se haya seleccionado el que se va a codificar. Haciendo clic en el icono *Devices*, en la parte inferior izquierda del panel de navegación, y a continuación, cuando desplace el puntero sobre el nombre del dispositivo, el símbolo de la estrella aparecerá a la izquierda, tal y como se muestra en la Figura 13.

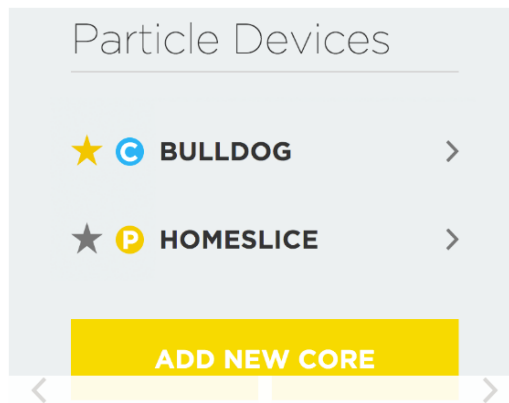


Figura 13. Particle Devices

- Hay que hacer clic en él para configurar el dispositivo que se va a actualizar (no estará visible si solo hay un único dispositivo). Una vez que haya seleccionado el dispositivo, la estrella asociada con él se pondrá amarilla.
- Barra de estado: Muestra información sobre el dispositivo seleccionado actualmente en la esquina inferior derecha del IDE web. Contiene lo siguiente: Nombre del último evento, Datos del último evento, Tipo de dispositivo, Nombre del dispositivo, Estado del dispositivo, Versión del dispositivo. Hacer clic en la bombilla indicará el dispositivo. En la Figura 14 se detalla lo descrito anteriormente.

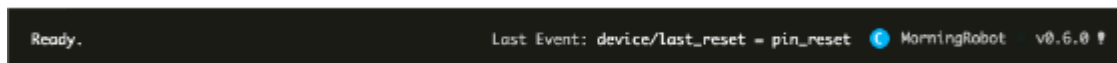


Figura 14. Barra de estado

- Flash*: haciendo clic en el botón *Flash* hará que el código se envíe de manera inalámbrica al dispositivo. Si el *flasheo* fue exitoso, el LED del dispositivo comenzará a parpadear en magenta.

3.2 Particle Cloud

La API de *Particle Cloud* es una REST (*Representational State Transfer*) API, este tipo de API utiliza solicitudes HTTP para obtener datos GET, POST, PUT y DELETE. Por tanto, Particle Cloud permite controlar diferentes dispositivos a través de Internet, mediante la asignación de una URL única a cada dispositivo [26]. Las variables y las funciones escritas en el firmware están expuestas como sub-recursos en el dispositivo. Todas las solicitudes del dispositivo llegan a través del servidor API utilizando seguridad TLS (*Transport Layer Security*).

El uso de Internet facilita la creación de dispositivos interactivos conectados, permitiendo que la información de los componentes esté disponible en línea con una línea de código adicional. Gracias a Internet, se consigue informar a otros dispositivos sobre lo que está sucediendo, e incluso se puede hacer que respondan. Además, facilita la conexión a las aplicaciones en línea para obtener información en tiempo real sobre el mundo, o decirle al mundo lo que está sucediendo con el dispositivo o en el área que lo rodea. La información que se desea compartir se presenta de una manera organizada para facilitar que otros dispositivos, servicios, aplicaciones y herramientas se conecten a ella. Lo hace como una API estructurada o una interfaz de programación de aplicaciones.

La API de *Particle Cloud* acepta solicitudes en formato JSON (*JavaScript Object Notation*) y en formato codificado, aunque siempre responde con JSON. Se utiliza el formato JSON porque cuando se intercambian datos entre un navegador y un servidor, los datos solo pueden ser de texto. JSON es texto, y se puede convertir cualquier objeto de *JavaScript* en JSON, y enviar JSON al servidor. También se puede convertir cualquier JSON recibido del servidor en objetos *JavaScript*. De esta manera se puede trabajar con los datos como objetos de *JavaScript*, sin análisis y traducciones complicadas.

Por otro lado, el formato codificado no es más que una solicitud de tipo POST. POST es un método de solicitud soportado por HTTP, usado por la WWW. Por diseño, este método solicita que un servidor web acepte los datos incluidos en el cuerpo del mensaje de solicitud, para almacenarlos o devolver algún tipo de información. Como parte de una solicitud POST, se puede enviar una cantidad arbitraria de datos de cualquier tipo al servidor en el cuerpo del mensaje de solicitud.

El hecho de haber conectado el dispositivo a Internet no significa que nadie más deba tener acceso a él, por eso *Particle Cloud* ofrece una serie de permisos para controlar y comunicar con el dispositivo. Estos permisos se gestionan con *OAuth2* que es el protocolo estándar de la industria para la autorización. Este protocolo se enfoca en la simplicidad de desarrollo del cliente y proporciona flujos de autorización específicos para aplicaciones web, aplicaciones de escritorio, teléfonos móviles y dispositivos de salón.

4. Alexa

La principal diferencia entre Alexa y el resto de asistentes virtuales es que Alexa tiene forma física, por lo que se puede colocar en cualquier lugar de la casa, y está siempre dispuesta a escuchar las órdenes de los habitantes del hogar a través de los micrófonos integrados que posee el dispositivo *Amazon Echo*. El invento de Amazon tiene forma de cilindro y cada vez que escuche su nombre, Alexa, intentará obedecer a todo lo que se le pida utilizando sus *skills* [27].

Para poder asimilar este nuevo concepto de una forma sencilla, se puede decir que las *skills* de Alexa son como las aplicaciones que se descargan en los *smartphones*. A través de la aplicación de Alexa o una web, se habilitan o deshabilitan dichas *skills* otorgando diferentes funcionalidades a Alexa. Al igual que a día de hoy se encuentran en Apple Store o en Google Play una gran variedad de aplicaciones con las que poder realizar casi cualquier tipo de actividad, actualmente ya existen *skills* de recetas, juegos, domótica, música, etc. Aunque dentro de las más utilizadas se encuentran las que permiten activar algunos componentes del hogar como la calefacción, la televisión, e incluso el robot roomba. Actualmente, según datos de Voicebot, en inglés ya hay más de 30.000 *skills* activas. Tal y como se observa en la Figura 15, toda *skill* de Alexa está compuesta de una *skill service* y una *skill interface*.

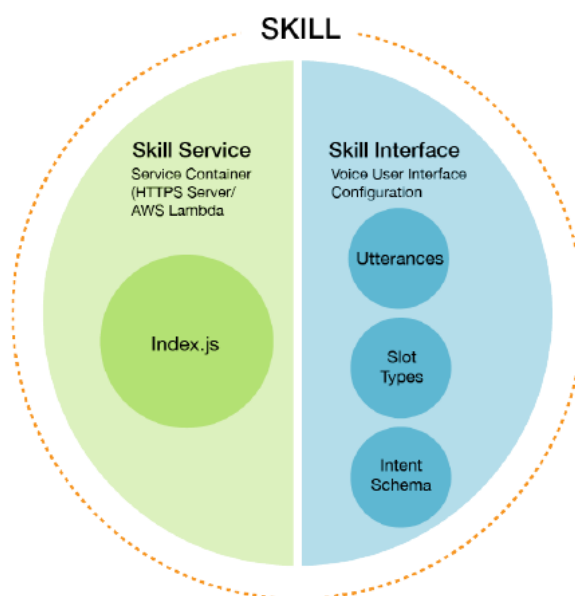


Figura 15. Alexa Skills

4.1 Skill Interface

La *Skill Interface* es la parte de la *skill* de Alexa que se encarga de la interacción con el usuario por medio de la voz. Por este motivo, cuando se diseña una *skill* de Alexa, se debe dedicar un poco de

tiempo a pensar acerca de cómo el usuario va a interactuar con la *skill* por medio de la voz. Los principales conceptos a tener en cuenta son los siguientes:

- *Invocation name*

El *invocation name* identifica los comandos que se utilizarán para iniciar la *skill*. Por ejemplo, el dispositivo *Amazon Echo* se activará con la palabra *Alexa* o *Amazon* o *Echo*, de este modo la interacción más común vendría a tener la siguiente forma:

Alexa, ask <invocation name> to <Intent>

- *Intents*

El siguiente concepto en el que enfocarse vendrían a ser los *intents*. Un *intent* es una descripción de lo que el usuario desea lograr al invocar la *skill* de *Alexa*. Por ejemplo, para crear una *skill* que responda a las peticiones del estado actual del LED del dispositivo *Photon*, hay que incluir en un archivo *.json* lo siguiente:

```
{
  "intents": [
    {
      "intent": "ParticleStateIntent",
    }
  ]
}
```

La habilidad de *Alexa* puede tener uno o más *intents*. Los *intents* se definen básicamente de este modo por su nombre, siendo esta la forma más simple de definirlos ya que en el ejemplo anterior solo se invoca el comando y no se le pasa ninguna variable o valor junto a este.

- *Slot*

El *Intent Schema* es muy flexible y también puede incluir parámetros, los cuales se definirían mediante *slots*. Los *slots* representan la información variable de lo que el usuario desea lograr. Por ejemplo, en este caso el estado del LED puede ser encendido (*on*) o apagado (*off*). Los *slots* se definen dentro del *Intent Schema* de la siguiente manera:

```
{
  "intents": [
    {
      "intent": "ParticleStateIntent",
```

```

        "slots": [
            {
                "name": "state",
                "type": "LIST_OF_STATES"
            }
        ]
    }
}

```

Los *slots* se diferenciarán entre sí por un nombre. En este caso, el nombre del slot es “*state*”, además, como un mismo slot puede tener diferentes variables, éstas se diferenciarán entre sí al incluirlas en un archivo .txt. En el *Intent Schema* se debe incluir el nombre del archivo asociado a dicho slot, tal y como se refleja en el ejemplo anterior. El contenido del archivo es el siguiente:

onoff

- Utterances

Los *utterances* no son más que una lista que especifica los diferentes comandos de voz que pueden utilizarse para referirse a un mismo *intent*. Estos se deben incluir en un archivo .txt y se describen de la siguiente manera:

ParticleStateIntent *turn* {*state*}

Donde *ParticleStateIntent* hace referencia al *intent* en cuestión, y lo que se sitúa entre corchetes hace referencia al slot. Esto combinado con el texto comprendido entre ellos define los comandos de voz que se pueden utilizar en cada caso.

4.2 Skill Service

La *Skill Service* de Alexa representa la parte en la que se realiza la comunicación de los dispositivos con los servidores de Amazon. Para que la comunicación sea efectiva hay que subir un fichero .ZIP a *AWS Lambda Setup*, que contenga el archivo *AlexaSkill.js* que proporcionará un conjunto básico de funcionalidades para la *skill* que se vaya a desarrollar, el archivo .js en el cual se implementará el *skill service*, y los paquetes *Node.js* que sean necesarios para el correcto funcionamiento del *skill service*. En las siguientes figuras se detalla el funcionamiento de *AlexaSkill.js*.

```

11  'use strict';
12
13  function AlexaSkill(appId) {
14    this._appId = appId;
15  }
16
17  AlexaSkill.prototype.speechOutputType = {
18    PLAIN_TEXT: 'PlainText',
19    SSML: 'SSML'
20  };

```

Figura 16. *AlexaSkill.js (I)*

En la línea 11 de la Figura 16 se encuentra el modo *'use strict'*, el cual se incluye para evitar el uso de variables no declaradas. La función *AlexaSkill* se encarga de que las comunicaciones entre los dispositivos y el servidor se realicen de forma segura, administrando la gestión de los mecanismos de solicitudes y respuestas. En las líneas 13-15 se declara esta función y se le pasa por parámetros *appId* para que pueda gestionar las solicitudes entrantes de verificación de la aplicación. En el caso de que las *appId* no coincidan, *AlexaSkill* no podrá llamar a ningún método.

Las líneas 17-20 declaran de qué tipo serán las propiedades del objeto *speechOutput*. Este objeto se encarga de guardar el discurso que se va a enviar al usuario como respuesta. *PlainText* indica que la salida del discurso está definida como un texto sin formato, mientras que *SSML* es el tipo de lenguaje que se utilizará para convertir el texto sin formato en voz sintética.

```

22  AlexaSkill.prototype.requestHandlers = {
23    LaunchRequest: function(event, context, response) {
24      this.eventHandlers.onLaunch.call(this, event.request, event.session, response);
25    },
26
27    IntentRequest: function(event, context, response) {
28      this.eventHandlers.onIntent.call(this, event.request, event.session, response);
29    },
30
31    SessionEndedRequest: function(event, context) {
32      this.eventHandlers.onSessionEnded(event.request, event.session);
33      context.succeed();
34    }
35  };

```

Figura 17. *AlexaSkill.js (II)*

Las líneas 22-35 de la Figura 17, describen la manera en la que la función *AlexaSkill* va a gestionar los diferentes tipos de solicitud. A pesar de que existen más tipos de solicitudes, esta función se centra en la gestión de tres:

- *LaunchRequest*: se utiliza cuando el usuario invoca la *skill* sin proporcionar un *intent* específico.
- *IntentRequest*: se envía cuando el usuario realiza una solicitud que corresponde a uno de los *intent* definidos en el *IntentSchema*.
- *SessionEndedRequest*: se activa cuando finaliza la sesión del *skill service* actual por cualquier motivo que no sea el código que cierra la sesión.

Para gestionar cada una de estas solicitudes se utiliza una función específica para cada una de ellas, y ésta a su vez se apoya en la función *eventHandlers*, la cual está definida más abajo en el código, y que es la encargada de lidiar con los parámetros para poder llevar a cabo cada solicitud.

Los parámetros son los siguientes:

- *Event*: Indica que tipo de método se va a utilizar.
- *Context*: Proporciona a la *skill* información sobre el estado actual del servicio y el dispositivo de Alexa en el momento en que se envía la solicitud.
- *Response*: Alexa le envía una solicitud a la *skill service* utilizando uno de los tipos de solicitud estándar cuando el usuario se comunica con la *skill* por voz. Los tipos de solicitud utilizados son los mencionados anteriormente.

```

40 AlexaSkill.prototype.eventHandlers = {
41   /**
42    * Called when the session starts.
43    * Subclasses could have overridden this function to open any necessary resources.
44    */
45   onSessionStarted: function(sessionStartedRequest, session) {},
46
47   /**
48    * Called when the user invokes the skill without specifying what they want.
49    * The subclass must override this function and provide feedback to the user.
50    */
51   onLaunch: function(launchRequest, session, response) {
52     throw 'onLaunch should be overridden by subclass';
53   },
54
55   /**
56    * Called when the user specifies an intent.
57    */
58   onIntent: function(intentRequest, session, response) {
59     var intent = intentRequest.intent,
60         intentName = intentRequest.intent.name,
61         intentHandler = this.intentHandlers[intentName];
62     if (intentHandler) {
63       console.log('dispatch intent = ' + intentName);
64       intentHandler.call(this, intent, session, response);
65     } else {
66       throw 'Unsupported intent = ' + intentName;
67     }
68   },
69
70   /**
71    * Called when the user ends the session.
72    * Subclasses could have overridden this function to close any open resources.
73    */
74   onSessionEnded: function(sessionEndedRequest, session) {}
75 };

```

Figura 18. *AlexaSkill.js (III)*

En la Figura 18 se encuentran las líneas 40-75 de *AlexaSkill.js*, en las que se define la función *EventHandlers*. Esta función se encarga de invocar el método de la clase *Speechlet* adecuado para cada solicitud. Los métodos son los siguientes:

- *OnSessionStarted*: Se llama cuando se inicia una sesión.
- *OnLaunch*: Llamado cuando el usuario invoca una *skill* sin especificar lo que quiere que haga.
- *OnIntent*: Utilizado cuando el usuario invoca un *intent*.
- *OnSessionEnded*: Se llama cuando se cierra una sesión.

La línea 80 de la Figura 19 define la función *intentHandlers*, la cual se deberá sobrescribir en el fichero *index.js* para especificar su funcionalidad. Esta función es la encargada de que se ejecuten

las diferentes peticiones del usuario mediante la comunicación con los servidores de Amazon. Las líneas 83-91 son las encargadas del proceso de verificación, en las líneas 93-95 se establecen los valores iniciales de los parámetros *sesión*. La invocación del método adecuado cuando se inicia una sesión se realiza en las líneas 97-99 y en las líneas 101-108 se envía la solicitud al controlador adecuado, teniendo en cuenta que éste puede haber sido sobrescrito.

```
80  AlexaSkill.prototype.intentHandlers = {};  
81  
82  AlexaSkill.prototype.execute = function(event, context) {  
83    try {  
84      console.log('session applicationId: ' + event.session.application.applicationId);  
85  
86      // Validate that this request originated from authorized source.  
87      if (this._appId && event.session.application.applicationId !== this._appId) {  
88        console.log('The applicationIds don\'t match : ' + event.session.application.applicationId + ' and '  
89          + this._appId);  
90        throw 'Invalid applicationId';  
91      }  
92  
93      if (!event.session.attributes) {  
94        event.session.attributes = {};  
95      }  
96  
97      if (event.session.new) {  
98        this.eventHandlers.onSessionStarted(event.request, event.session);  
99      }  
100  
101     // Route the request to the proper handler which may have been overridden.  
102     var requestHandler = this.requestHandlers[event.request.type];  
103     requestHandler.call(this, event, context, new Response(context, event.session));  
104   } catch (e) {  
105     console.log('Unexpected exception ' + e);  
106     context.fail(e);  
107   }  
108   };
```

Figura 19. AlexaSkill.js (IV)

En la Figura 20 se definen un objeto y una función. El objeto (líneas 109-112) se utiliza para realizar las comunicaciones con Alexa, y sus dos parámetros contienen la información necesaria para que el servicio realice su lógica y genere una respuesta con formato JSON. La segunda función (líneas 114-126) devolverá la información referente a los parámetros que tenga el parámetro del objeto *response* seleccionado.

```

109 var Response = function(context, session) {
110     this._context = context;
111     this._session = session;
112 };
113
114 function createSpeechObject(optionsParam) {
115     if (optionsParam && optionsParam.type === 'SSML') {
116         return {
117             type: optionsParam.type,
118             ssm1: optionsParam.speech
119         };
120     } else {
121         return {
122             type: optionsParam.type || 'PlainText',
123             text: optionsParam.speech || optionsParam
124         };
125     }
126 }

```

Figura 20. AlexaSkill.js (V)

```

128 Response.prototype = (function() {
129     var buildSpeechletResponse = function(options) {
130         var alexaResponse = {
131             outputSpeech: createSpeechObject(options.output),
132             shouldEndSession: options.shouldEndSession
133         };
134         if (options.reprompt) {
135             alexaResponse.reprompt = {
136                 outputSpeech: createSpeechObject(options.reprompt)
137             };
138         }
139         if (options.cardTitle && options.cardContent) {
140             alexaResponse.card = {
141                 type: 'Simple',
142                 title: options.cardTitle,
143                 content: options.cardContent
144             };
145         }
146         var returnResult = {
147             version: '1.0',
148             response: alexaResponse
149         };
150         if (options.session && options.session.attributes) {
151             returnResult.sessionAttributes = options.session.attributes;
152         }
153         return returnResult;
154     };

```

Figura 21. AlexaSkill.js (VI)

Las líneas 128-154 de la Figura 21 representan un método del objeto *response* que se va a encargar de generar las diferentes respuestas que Alexa proporcionará al usuario. Para ello, posee una función llamada *buildSpeechletResponse* que será la encargada de crear los diferentes parámetros del objeto *response*. Esta función posee un objeto denominado *alexaResponse* que, dependiendo del número de parámetros que tenga la función que lo contiene en su entrada,

formará respuestas de diferentes tipos. Esta función debe devolver la versión y el resultado del objeto que ha creado.

En la Figura 22, las líneas 156-193 recogen los diferentes tipos de objetos *response* que se pueden crear en función de los objetos que tenga como parámetros. La línea 194 permite utilizar estas funciones en otros archivos.

```
156     return {
157         tell: function(speechOutput) {
158             this._context.succeed(buildSpeechletResponse({
159                 session: this._session,
160                 output: speechOutput,
161                 shouldEndSession: true
162             }));
163         },
164         tellWithCard: function(speechOutput, cardTitle, cardContent) {
165             this._context.succeed(buildSpeechletResponse({
166                 session: this._session,
167                 output: speechOutput,
168                 cardTitle: cardTitle,
169                 cardContent: cardContent,
170                 shouldEndSession: true
171             }));
172         },
173         ask: function(speechOutput, repromptSpeech) {
174             this._context.succeed(buildSpeechletResponse({
175                 session: this._session,
176                 output: speechOutput,
177                 reprompt: repromptSpeech,
178                 shouldEndSession: false
179             }));
180         },
181         askWithCard: function(speechOutput, repromptSpeech, cardTitle, cardContent) {
182             this._context.succeed(buildSpeechletResponse({
183                 session: this._session,
184                 output: speechOutput,
185                 reprompt: repromptSpeech,
186                 cardTitle: cardTitle,
187                 cardContent: cardContent,
188                 shouldEndSession: false
189             }));
190         }
191     };
192     }());
193
194     module.exports = AlexaSkill;
```

Figura 22. *AlexaSkill.js* (VII)

4.3 Test

A medida que se avanza en el desarrollo de un proyecto, es normal encontrarse con algunos fallos en el código elaborado. Existen diferentes técnicas para la detección y eliminación de estos fallos, aunque estas pueden convertirse en una tarea bastante tediosa si no se sabe con exactitud en que parte del código se están produciendo los errores.

Alexa ofrece una herramienta con la que realizar test a las funciones creadas dentro de su entorno. Esta herramienta facilita la detección y eliminación de código, ya que es capaz de verificar que el funcionamiento de las *skills* es correcto. La verificación del correcto funcionamiento de las *skills* permite descartar esta parte del código en la búsqueda de errores.

Para poder utilizar esta herramienta es necesario dirigirse a la consola de *AWS Lambda* y seleccionar la función que se desea testear, seleccionar el tipo de test a realizar, y presionar en el botón probar. A continuación, se describirá el funcionamiento de un test que se encarga de verificar el cambio de estado del slot de una *skill* de Alexa. Las líneas 1-10 de la Figura 23, verifican que se cumplen los requisitos necesarios para establecer una nueva sesión con la *skill* de Alexa. Las líneas 13-28 indican el tipo de petición que se va a realizar para verificar el funcionamiento de la *skill*, en este caso se va a cambiar el estado inicial del *intent ParticleStateIntent* del slot *off* al *slot on*. La expresión *[unique-value-here]* que aparece en el código indica que esa parte del código debe ser sustituida por el *Skill ID*, el cual se obtiene de la consola *Alexa Developer* y se explicará más adelante en el desarrollo de la memoria.

```
1 {
2   "session": {
3     "new": false,
4     "sessionId": "amzn1.echo-api.session.[unique-value-here]",
5     "attributes": {},
6     "user": {
7       "userId": "amzn1.ask.account.[unique-value-here]"
8     },
9     "application": {
10      "applicationId": "amzn1.ask.skill.[unique-value-here]"
11    }
12  },
13  "version": "1.0",
14  "request": {
15    "locale": "en-US",
16    "timestamp": "2016-10-27T21:06:28Z",
17    "type": "IntentRequest",
18    "requestId": "amzn1.echo-api.request.[unique-value-here]",
19    "intent": {
20      "slots": {
21        "state": {
22          "name": "state",
23          "value": "on"
24        }
25      },
26      "name": "ParticleStateIntent"
27    }
28  },
```

Figura 23. Probando Test (I)

Las líneas 29-47 de la Figura 24, indican que se administrará y monitoreará la progresión de la reproducción de voz y eventos por medio de la interfaz *AudioPlayer*. Esta interfaz verificará que las directivas enviadas por la *skill* para enviar y detener la reproducción son las correctas.

```
29 - "context": {
30 -   "AudioPlayer": {
31 -     "playerActivity": "IDLE"
32 -   },
33 -   "System": {
34 -     "device": {
35 -       "supportedInterfaces": {
36 -         "AudioPlayer": {}
37 -       }
38 -     },
39 -     "application": {
40 -       "applicationId": "amzn1.ask.skill.[unique-value-here]"
41 -     },
42 -     "user": {
43 -       "userId": "amzn1.ask.account.[unique-value-here]"
44 -     }
45 -   }
46 - }
47 - }
```

Figura 24. Probando Test (II)

5. Integración del dispositivo Photon con el software de Alexa

5.1 Configuración de la Skill Interface y Skill Service de Alexa

Para afianzar los conceptos de Alexa se decidió realizar una pequeña aproximación al objetivo general del presente TFG, que consistía en encender un LED conectado al dispositivo *Photon* por medio de Alexa. Este LED puede ser el que trae incorporado el *Photon* o uno que se ha conectado a uno de sus pines, tal y como se muestra en la Figura 25. Alexa debe ser capaz de encender uno u otro en función de las peticiones del usuario.

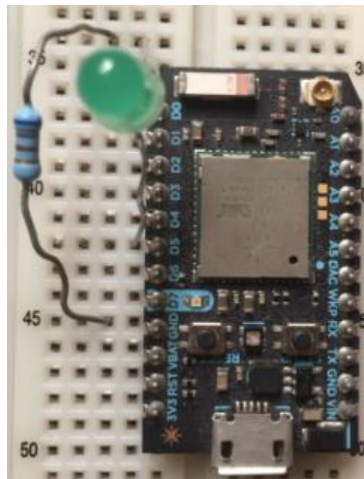


Figura 25. Configuración del hardware del Photon

Para que Alexa sea capaz de gestionar las peticiones del usuario es necesario describir la *skill service*. Lo primero es definir el *invocation name*, que en este caso será *ParticleLight*. La mejor manera de conseguir que Alexa fuera capaz de encender el LED adecuado es mediante el uso de dos *intent*, la Figura 26 recoge el *Intent Schema*:

```
{
  "intents": [
    {
      "intent": "ParticleStateIntent",
      "slots": [
        {
          "name": "state",
          "type": "LIST_OF_STATES"
        }
      ]
    },
    {
      "intent": "ParticleSelectIntent",
      "slots": [
        {
          "name": "select",
          "type": "LIST_OF_SELECTIONS"
        }
      ]
    }
  ]
}
```

Figura 26. Intent Schema Photon

El *intent ParticleStateIntent* tiene un slot llamado *state* que puede tener dos estados: *on* y *off*. Por otro lado, el *intent ParticleSelectIntent* tiene un *slot* denominado *select* y tiene dos estados: *Five* y *Seven*.

Solo se logrará integrar el dispositivo *Photon* con Alexa si la configuración de la *skill service* se ha realizado por completo, para lo cual hay que completar el archivo *.js*. A continuación, se va a describir el código de este archivo por medio de imágenes. La línea 4 de la Figura 27 representa la variable que guardará el identificador de la *skill* de Alexa, en este caso aún no está definido ya que más adelante se explicará cómo se obtiene.

Para llevar a cabo este proyecto con dispositivos soportados por Particle, *Photon* en este caso, se necesita hacer uso de las herramientas que facilita Particle para la transmisión de datos entre sus dispositivos y servidores. Por este motivo esta empresa se ha encargado de desarrollar una librería *JavaScript* que sea capaz de gestionar la API de *Particle Device Cloud* utilizando *Node.js*. En la línea 10 se exporta un módulo de esta librería a la variable *Particle* a través de la función *require()*, mientras que en la línea 11 se crea un objeto para acceder a todas las funciones de este módulo. La línea 17 es la variable en la que se va a guardar el identificador del dispositivo, este identificador se obtiene en Particle Web IDE en el apartado *devices*, haciendo clic sobre el dispositivo que se va a utilizar. La línea 18 es una variable que guarda el valor del *Access Token*, este se obtiene en Particle Web IDE en el apartado *Settings*.

En la línea 23 se exporta el módulo *Alexaskill* del archivo *AlexaSkill.js*. Las líneas 31-32 representan la declaración de una clase que hereda directamente de *AlexaSkill.js* y, por tanto, será la encargada de definir como la *skill service* gestiona las llamadas de la *skill* interface. La línea 36 le proporciona a esta clase la habilidad para heredar propiedades de la clase *AlexaSkill* del archivo *AlexaSkill.js*, mientras que la línea 37 es la declaración del constructor de esta clase.


```

1  /**
2   * App ID for the skill
3   */
4   var APP_ID = undefined;
5
6  /**
7   * Use Particle API JS to access the temperature / humidity data
8   * @see https://docs.particle.io/reference/javascript/
9   */
10  var Particle = require('particle-api-js');
11  var particle = new Particle();
12
13  /**
14   * Particle authentication credentials
15   * @see https://docs.particle.io/reference/api/#authentication
16   */
17  var PARTICLE_DEVICE_ID = "350041000d47343233323032";
18  var PARTICLE_ACCESS_TOKEN = "ec56a85fc13875cdacd2055a149a7d5f9f16203b";
19
20  /**
21   * The AlexaSkill prototype and helper functions
22   */
23  var AlexaSkill = require('./AlexaSkill');
24
25  /**
26   * ParticleSkill is a child of AlexaSkill.
27   * To read more about inheritance in JavaScript, see the link below.
28   *
29   * @see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript#Inheritance
30   */
31  var ParticleSkill = function () {
32    AlexaSkill.call(this, APP_ID);
33  };
34
35  // Extend AlexaSkill
36  ParticleSkill.prototype = Object.create(AlexaSkill.prototype);
37  ParticleSkill.prototype.constructor = ParticleSkill;

```

Figura 27. Index.js Photon (I)

La Figura 28 describe la parte del código en la que se sobrescriben las funciones *OnSessionStarted*, *OnLaunch* y *OnSessionEnded* del archivo *AlexaSkill.js* para añadirles la funcionalidad deseada de acuerdo a las características de cada función. Principalmente se les indica el tipo de solicitud que van a gestionar y en el caso de la función *OnLaunch*, también se le indica el tipo de respuesta que debe dar.

```

39  ParticleSkill.prototype.eventHandlers.onSessionStarted = function (sessionStartedRequest, session) {
40    console.log("ParticleSkill onSessionStarted requestId: " + sessionStartedRequest.requestId
41      + ", sessionId: " + session.sessionId);
42    // any initialization logic goes here
43  };
44
45  ParticleSkill.prototype.eventHandlers.onLaunch = function (launchRequest, session, response) {
46    console.log("ParticleSkill onLaunch requestId: " + launchRequest.requestId + ", sessionId: " + session.sessionId);
47    var speechOutput = "Welcome to Particle Light. You can ask me to turn on or off the light you have selected.";
48    var repromptText = "You can ask me to turn the light on or off.";
49    response.ask(speechOutput, repromptText);
50  };
51
52  ParticleSkill.prototype.eventHandlers.onSessionEnded = function (sessionEndedRequest, session) {
53    console.log("ParticleSkill onSessionEnded requestId: " + sessionEndedRequest.requestId
54      + ", sessionId: " + session.sessionId);
55    // any cleanup logic goes here
56  };

```

Figura 28. Index.js Photon (II)

En la Figura 29 se sobrescribe la función *intentHandlers* de *AlexaSkill.js*. En ella se describe de qué manera se va a realizar la gestión de los *intent* cuando un usuario los invoca. Para el caso del *intent ParticleStateIntent*, éste se va a gestionar por medio de una función que tiene como parámetros el *intent*, la sesión, y el tipo de respuesta. La línea 61 representa una variable en la que se va a guardar el nombre del *slot*. La línea 62 contiene una variable cuya declaración va a

depender de si existe la variable *slot*. En caso de existir esta variable, guardará el estado del *slot* y en caso contrario, guardará una cadena vacía.

Las líneas 64-71 contienen una sentencia *if* que solo se va a cumplir si la variable *SlotValue* anteriormente descrita no contiene una cadena vacía. En este caso, se creará una clase que llame a la función de Particle correspondiente, para lo cual se le debe indicar el identificador del dispositivo, el nombre y el estado del *slot*, y el *Access token*. Las líneas 72-81 determinan si la llamada de la función de Particle ha sido satisfactoria o no. En caso de ser satisfactoria envía la respuesta que Alexa mandará al usuario, y en caso contrario, avisa de que ha habido un error. Las líneas 82-85 solo se ejecutarán si la sentencia *if* no se ha cumplido, y Alexa indicará con su respuesta al usuario que se ha producido un error.

```
58 ParticleSkill.prototype.intentHandlers = {
59   // register custom intent handlers
60   "ParticleStateIntent": function (intent, session, response) {
61     var slot = intent.slots.state;
62     var slotValue = slot ? slot.value : "";
63
64     if(slotValue) {
65       var fnPr = particle.callFunction({
66         deviceId: PARTICLE_DEVICE_ID,
67         name: slot.name,
68         argument: slotValue,
69         auth: PARTICLE_ACCESS_TOKEN
70       });
71
72       fnPr.then(
73         function(data) {
74           console.log('Function called succesfully:', data);
75
76           var speechOutput = "The light is now " + slotValue;
77           response.tellWithCard(speechOutput, "Particle Light", speechOutput);
78         }, function(err) {
79           console.log('An error occurred:', err);
80         });
81     }
82     else {
83       response.tell("Sorry, I didn't catch what you said");
84     }
85   },
```

Figura 29. Index.js Photon (III)

La Figura 30 representa el código de la gestión del *intent ParticleSelectIntent*, las diferencias entre este código y el código del *intent ParticleStateIntent* radican en el tipo de respuesta que Alexa proporciona al usuario en caso de haber llevado a cabo correctamente la llamada a la función de Particle. Por otro lado, en las líneas 115-119 se crea el objeto que invoca *AWS Lambda*. *AWS Lambda* invoca su función *Lambda* a través de un objeto *handler*. Un *handler* representa el nombre de la función *Lambda*, y es el punto de entrada que *AWS Lambda* utiliza para ejecutar el código de su función.

```

86 ▾ "ParticleSelectIntent": function (intent, session, response) {
87     var slot = intent.slots.select;
88     var slotValue = slot ? slot.value : "";
89
90 ▾     if(slotValue) {
91 ▾         var fnPr = particle.callFunction({
92             deviceId: PARTICLE_DEVICE_ID,
93             name: slot.name,
94             argument: slotValue,
95             auth: PARTICLE_ACCESS_TOKEN
96         });
97
98         fnPr.then(
99 ▾             function(data) {
100                 console.log('Function called succesfully:', data);
101
102                 var speechOutput = "The selected light is now " + slotValue;
103                 response.tellWithCard(speechOutput, "Particle Light", speechOutput);
104 ▾             }, function(err) {
105                 console.log('An error occurred:', err);
106             });
107         }.
108         else {
109             response.tell("Sorry, I didn't catch what you said");
110         }.
111     }.
112 };
113
114 // Create the handler that responds to the Alexa Request.
115 ▾ exports.handler = function (event, context) {
116     // Create an instance of the ParticleSkill skill.
117     var particleSkill = new ParticleSkill();
118     particleSkill.execute(event, context);
119 };

```

Figura 30. Index.js Photon (IV)

Llegados a este punto, ya se conoce el funcionamiento del archivo *AlexaSkill.js* y se ha determinado el código del archivo *.js* que va a implementar la *skill service*. Para terminar de completar la *skill service* falta por añadir los paquetes *Node.js* utilizados en el archivo *.js*. En este caso, al archivo *.js* se le asignó el nombre de *index.js*, y éste junto, con el archivo *AlexaSkill.js* se guardaron dentro de una misma carpeta, llamada *src*. Por otro lado, la mejor forma de gestionar los paquetes *npm* instalados localmente es creando un archivo *.json* cuyo contenido debe ser similar al que aparece en la Figura 31:

```

{
  "name": "src",
  "version": "1.0.0",
  "dependencies": {
    "particle-api-js": "^7.2.3"
  }
}

```

Figura 31. package.json Photon

Los requisitos mínimos de cualquier archivo *.json* son el apartado *name* y *version*. *Name* puede ser sustituido por cualquier valor y *version* también siempre y cuando siga el formato x.x.x. El apartado *dependencies* indica los paquetes que se van a utilizar en el proyecto. En este caso hay que indicar el nombre del paquete y su versión con el formato *^x.x.x*, mientras que la versión se puede obtener directamente de la página *npm* en la que está guardado el paquete que se quiere instalar. Una vez que se ha creado el archivo *.json*, se guarda en la carpeta *src*. Para instalar el paquete *Node.js* basta con ejecutar en terminal el comando *npm install* dentro de la carpeta en la que están almacenados los archivos.

5.2 Configuración de los servicios de Amazon Alexa

En este punto ya se ha determinado todo el código que van a utilizar las *skills* de Alexa. Para que Alexa lo pueda interpretar hay que subirlo a los servicios de Amazon Alexa. El primer paso es configurar *AWS Lambda*, para ello hay que ingresar en la consola *AWS* de Amazon con una cuenta y acceder a la consola *Lambda* para crear una función desde cero, el nombre de la función será *particlelight*. Al crear la función *AWS* asigna inmediatamente al usuario un código *ARN*. Es importante guardar este código y acceder a las *skills* de *Alexa Developer Console*, ya que por el momento no se puede continuar con la configuración de *AWS Lambda* sin el *ID* de la *skill*.

En *Alexa Developer Console* se creará la *skill*, el nombre de la *skill* será *particlelight*. El modelo a seleccionar es el de *Start from Scratch*. El *invocation name* a utilizar es *particle light*. El siguiente paso es añadir los *intent*, *samples* y *slots*. Los *intent* serán *ParticleStateIntent* y *ParticleSelectIntent*, los *samples utterances* serán:

- *ParticleStateIntent state {state}*.
- *ParticleSelectIntent select {select}*.

Los *slots* serán *state* y *select*. Los *slots types* serán *LIST_OF_STATES*, cuyos estados serán *on* y *off*, y *LIST_OF_SELECT* cuyos estados serán *FIVE* y *SEVEN*. Una vez añadido en *Alexa Developer Console*, debe quedar tal y como se muestra en la Figura 32.

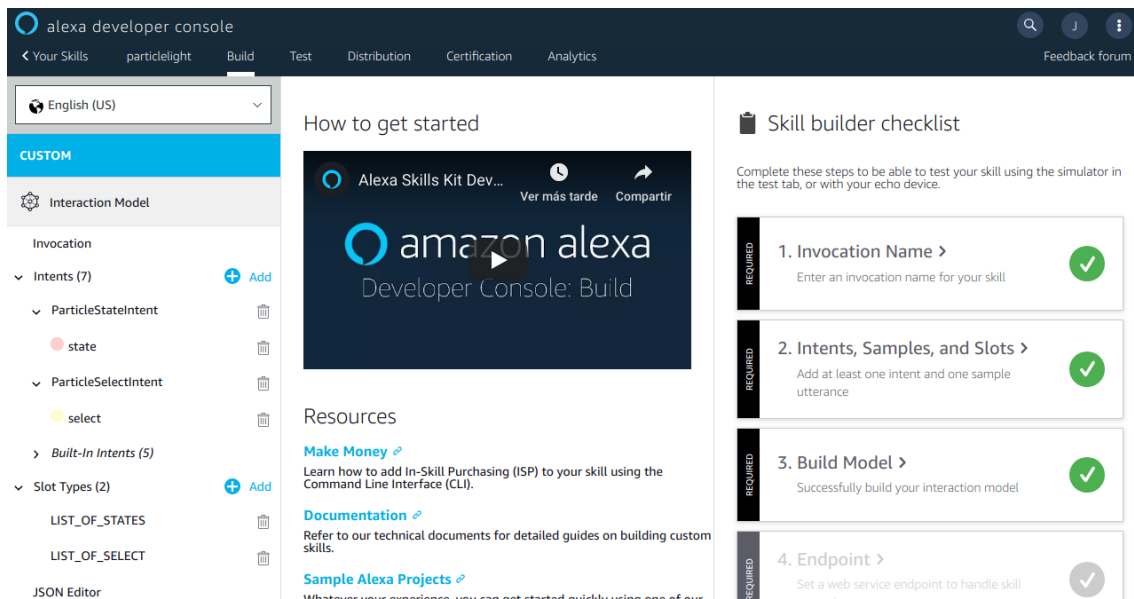


Figura 32. Alexa Developer Console

Para finalizar con la configuración de *Alexa Developer Console* hay que añadir un *endpoint*. En este caso, el *endpoint* debe ser del tipo *AWS Lambda ARN* y en el único apartado obligatorio de esa sección hay que añadir el *ARN* obtenido de *AWS Lambda*. Por otro lado, en esta misma sección se encuentra la *ID* de la *skill*, que hay que copiar e incluirla en la variable *APP_ID* del archivo *index.js*. Ahora que se ha obtenido la *ID* de la *skill*, se puede volver a *AWS Lambda* y elegir el desencadenador *Alexa Skill Kit*, al cual hay que añadirle el *ID* de la *skill* que se acaba de obtener.

A la función *AWS Lambda* creada anteriormente hay que añadirle un código. El tipo de entrada de este código será cargar un archivo *.zip*. Para ello hay que retomar la carpeta *src* y comprimir todos los archivos que hay en ella en un *.zip*, que posteriormente habrá que subir a *AWS Lambda*. En este momento, *AWS Lambda* debería verse como se refleja en la Figura 33.

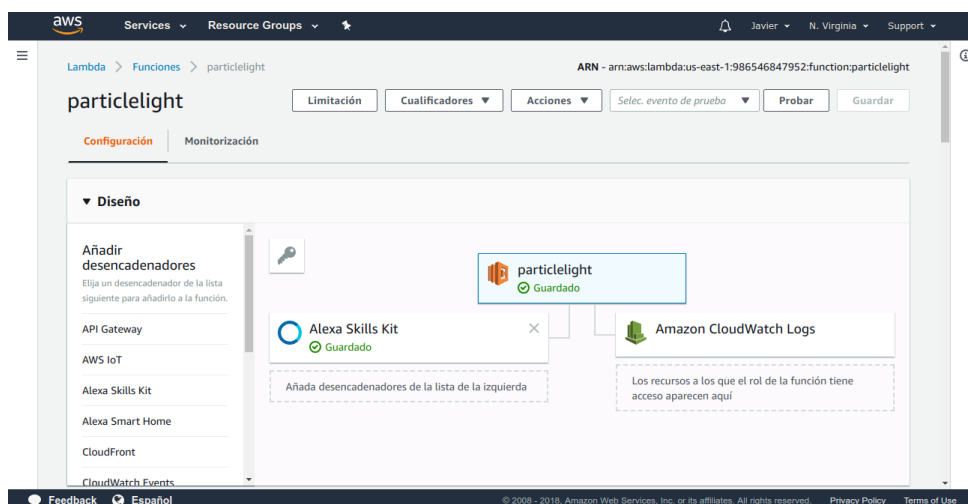


Figura 33. AWS Lambda

Ahora hay que comprobar mediante *test* que la *skill* funciona correctamente. Para ello se realizarán dos *tests*, uno en el que se cambie el estado del LED de su valor inicial *off* a *on* y otro en el que se cambie el LED seleccionado del valor inicial *FIVE* a *SEVEN*. El primer *test* es exactamente el mismo que se utilizó en el apartado 4.3, aunque en este se sustituyó *unique-value-here* por la *ID* de la *skill*. En la Figura 34 se puede comprobar como la simulación del *test* da como resultado lo esperado, cero errores.



Figura 34. Resultado test state intent

El segundo *test* se diferencia del primero en que el *intent* que se está verificando en este caso es diferente. La Figura 35 muestra estas diferencias.

```
1 {
2   "session": {
13   "version": "1.0",
14   "request": {
15     "locale": "en-US",
16     "timestamp": "2016-10-27T21:06:28Z",
17     "type": "IntentRequest",
18     "requestId": "amzn1.echo-api.request.20d014a5-5b9c-45b4-9c16-4dd8e02c060e",
19     "intent": {
20       "slots": {
21         "state": {
22           "name": "select",
23           "value": "FIVE"
24         }
25       }
26     },
27     "name": "ParticleSelectIntent"
28   },
29   "context": {
47 }
```

Figura 35. Test select intent

La Figura 36 demuestra que la simulación de este *test* también ha sido satisfactoria.



Figura 36. Resultado test select intent

5.3 Firmware de usuario del dispositivo *Photon*

Para finalizar la integración faltaría incluir en el *Photon* el código que enciende, apaga y selecciona el LED dependiendo de las funciones que haya llamado Alexa a través de Particle Cloud con la interacción por voz del usuario. El código del dispositivo *Photon* es el que se incluye en la Figura 37. Las líneas 1-2 asignan variables a la dirección de los pines, *D7* corresponde con el LED del *Photon* y *D5* corresponde al pin al que se ha conectado el nuevo LED. La línea 3 representa una variable en la que se va a guardar el valor del pin seleccionado. Las líneas 4-9 corresponden con la función *setup*, utilizada para registrar variables y funciones en la nube. La función *pinMode* configura el pin indicado como entrada o salida. En este caso ambos pines están configurados como salida. La función *Particle.function* permite que el código del dispositivo se ejecute cuando lo solicita Particle Cloud API. Tiene como parámetros *funckey*, que es el nombre del *string* que se utiliza para hacer realizar una solicitud POST, y *funcName*, que es el nombre de la función que recibe la llamada de la API.

Las líneas 12-37 concuerdan con la función *setLightState* que es la encargada de encender o apagar el LED seleccionado, en función del valor del *funckey* que se ha recibido. Las líneas 38-44 se ajustan a la función *selectLight* que se ocupa de seleccionar el LED que se va a encender o apagar dependiendo del *funckey* que reciba.

```

1  int led1 = D7;
2  int led2 = D5;
3  int selected;
4  void setup() {
5      pinMode(led1, OUTPUT);
6      pinMode(led2, OUTPUT);
7      Particle.function("state", setLightState);
8      Particle.function("select", selectLight);
9  }
10
11
12  int setLightState(String state) {
13  if (selected == 7){
14  if(state == "on") {
15      digitalWrite(led1, HIGH);
16  }
17  else if(state == "off") {
18      digitalWrite(led1, LOW);
19  }
20  }else if(selected == 5){
21  if(state == "on") {
22      digitalWrite(led2, HIGH);
23  }
24  else if(state == "off") {
25      digitalWrite(led2, LOW);
26  }
27  }else if (selected == 0){
28  if(state == "on") {
29      digitalWrite(led1, HIGH);
30  }
31  else if(state == "off") {
32      digitalWrite(led1, LOW);
33  }
34  }
35
36  }
37
38  int selectLight (String select){
39  if(select == "7"){
40      selected = 7;
41  }else if(select == "5"){
42      selected = 5;
43  }
44  }

```

Figura 37. Código software Photon

Ahora, ya se puede verificar el código en el Particle Web IDE en busca de errores. En caso de no tener errores, ya se puede *flashear* directamente en el dispositivo. Para comprobar que la integración del dispositivo *Photon* con Alexa se produjo de manera correcta, se recurrió a la herramienta Echosim.io, que sirve como herramienta de testeo de las *Alexa skills*. En la Figura 38 se muestra la interfaz de la página Echosim.io.

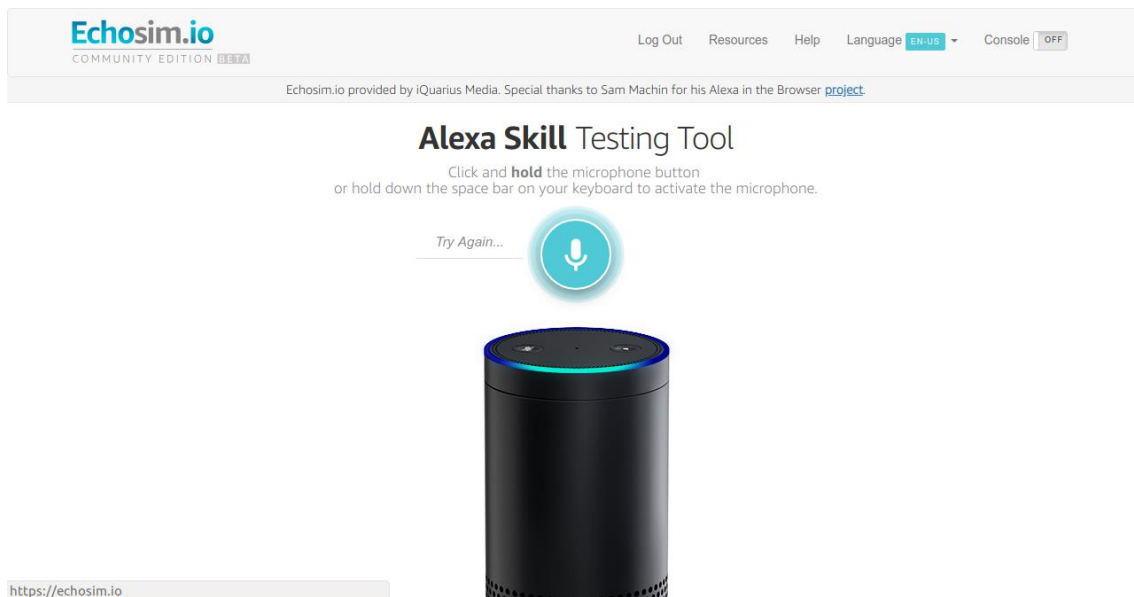


Figura 38. Echosim.io

Para poder testear la *skill* de Alexa basta con iniciar sesión con la cuenta de Amazon y, probar los siguientes comandos para ver que el funcionamiento es el esperado:

- *Alexa, open particle light*: comando para iniciar sesión.
- *Alexa, ask particle light to turn [on | off]*: comando para encender o apagar el LED seleccionado.
- *Alexa, ask particle light to select [FIVE | SEVEN]*: comando para seleccionar el LED que se va a encender o apagar.

6. Bluetooth Low Energy

6.1 Introducción

El objetivo principal de este TFG es conseguir la integración de Alexa con dispositivos BLE. Con este propósito, dada su incompatibilidad inicial, se utilizarán dispositivos IoT como pasarela. BLE se corresponde con la especificación 4.0 de la tecnología *Bluetooth* desarrollada por *Bluetooth SIG* en el año 2010, que es la que se considerará en este caso.

BLE se planteó como un estándar con objetivos y aplicaciones complementarias a las de *Bluetooth* convencional con las siguientes diferencias destacables: BLE es una tecnología orientada a garantizar un consumo de energía bajo, menor tiempo de establecimiento de conexión, y menor complejidad. BLE está diseñado para la transmisión de pequeñas cantidades de datos a muy corto alcance, con un reducido consumo de potencia. No está diseñado para establecer una conexión entre dispositivos que estén transmitiendo grandes cantidades de datos por un largo tiempo a alta velocidad, como en el caso de *Bluetooth* convencional.

Ambas tecnologías usan la banda ISM de 2.4 GHz. *Bluetooth* clásico y BLE usan la modulación GFSK a 1Mbps, pero con índices de modulación diferentes. El estándar *Bluetooth* clásico tiene 79 canales, mientras que BLE tiene 40, tal como se indica en la Tabla 2. La separación entre canales también es diferente. Debido a estas dos diferencias entre BLE y *Bluetooth* clásico, éstos son incompatibles entre sí. Sin embargo, existen dispositivos *Dual Mode* que soportan las dos tecnologías conmutando los parámetros de modulación y los canales donde se está emitiendo.

	BLE	Bluetooth Convencional
Modulación	GFSK (0.45 - 0.55)	GFSK (0.28 - 0.35)
Tasa Mbit/s	1 Mbit/s	1 Mbit/s
Número de canales	40	79
Separación	2 MHz	1 MHz

Tabla 2. Comparativa BLE y Bluetooth convencional

La banda ISM de 2.4 GHz ofrece una clara ventaja en términos de coste, ya que no es necesario pagar una licencia para su utilización. Sin embargo, al tratarse de una banda libre, surgen problemas de tráfico que pueden conllevar la congestión de la banda. De este problema surgió la necesidad de implementar un sistema capaz de trabajar con garantías en entornos con interferencias, para ello, se implementó la técnica *Adaptative Frequency Hopping*. Esta técnica permite detectar y evitar de forma adaptativa fuentes de interferencias. Además, la recuperación

efectiva de los paquetes perdidos durante la transmisión, hacen de esta tecnología *Bluetooth* un sistema de transmisión inalámbrica robusto. Trabajar en la banda de 2.4 GHz ofrece también la capacidad de operar a nivel global con un impacto mínimo en costes y facilidad para gestionar grandes volúmenes de facturación.

Hoy en día, el consumo de energía es importante para todos los proyectos, ya que este parámetro se refleja directamente en el coste de las infraestructuras. Del mismo modo, se requiere una vida larga de batería para la tecnología elegida con el objetivo de reducir el mantenimiento de los equipos. Por todo esto, la potencia de transmisión de este protocolo se establece en un rango comprendido entre -30 dBm y 0 dBm, rango que no representa un impacto significativo en el consumo de batería del dispositivo. Para incrementar la vida útil de la batería, se establece un rango de operación comprendido entre 2 y 5 metros de distancia.

6.2 Topología de red

BLE utiliza una topología de red de tipo estrella. Los dispositivos BLE pueden comunicarse de acuerdo a dos modos de operación:

- *Broadcasting*: La secuencia de envío de datos es en un solo sentido. Permite el envío de datos a cualquier dispositivo que se encuentre dentro del rango de escucha. En este modo de operación se definen dos roles distintos: *Broadcaster*, el cual envía paquetes de *Advertising* periódicamente a cualquier dispositivo BLE, y *Observer*, encargado de escanear repetidamente las frecuencias preconfiguradas para recibir cualquier paquete de *Advertising* que se esté emitiendo. Este modo de operación se resume en la Figura 39.

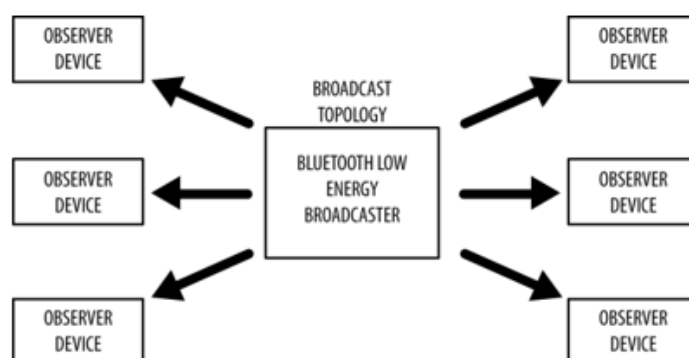


Figura 39. Modo de operación Broadcast [28].

- *Connections*: La secuencia de envío de datos se realiza en ambas direcciones. Este modo de operación implica un intercambio permanente y periódico de paquetes de datos entre dos dispositivos. Esta secuencia es exclusiva entre los dispositivos implicados en una

conexión. Dentro de esta topología existen dos roles diferentes. El primero, se denomina *Central*, el cual explora repetidamente las frecuencias preestablecidas para recibir paquetes de *Advertising* e iniciar una conexión cuando corresponda. Una vez que se establece la conexión, el dispositivo *Central* administra el timing e inicia los intercambios periódicos de datos. El otro rol es *Peripheral*, que se encarga de enviar paquetes de *Advertising* conectables periódicamente y aceptar las conexiones entrantes. Una vez que se encuentra en una conexión activa, el dispositivo *Peripheral* sigue la sincronización del dispositivo Central e intercambia datos regularmente con él.

En este modo de operación solo se inicia una conexión cuando el dispositivo Central recibe los paquetes de *Advertising* conectables. Solo en ese momento, el dispositivo *Central* será capaz de enviar una solicitud al dispositivo *Peripheral* para establecer una conexión privada entre ambos. Una vez que se establece la conexión, el dispositivo *Peripheral* deja de enviar paquetes de *Advertising* y los dos dispositivos pueden comenzar a intercambiar paquetes de datos en ambas direcciones, como se muestra en la Figura 40.

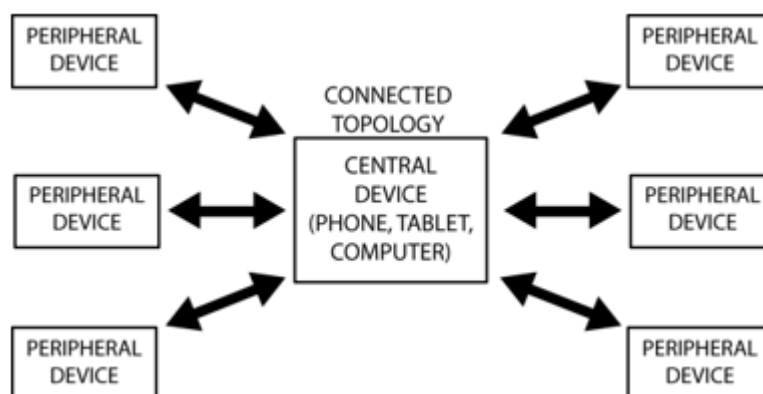


Figura 40. Modo de operación Connections. [28]

En este Trabajo Fin de Grado se hace uso del modo *Connections*, con lo que, se dispone de dos dispositivos que toman los roles de *Central* y *Peripheral*. En este caso, el dispositivo final BLE tomará las funciones de *Peripheral*, mientras que el dispositivo RedBear DUO realizará las funciones de Central.

6.3 Arquitectura

La arquitectura del estándar BLE viene definida por una pila de protocolos que se encarga de gestionar los dispositivos, la conexión y la interfaz de las aplicaciones. Esta pila se divide en tres grupos diferentes: *Application*, *Host* y *Controller*, los cuales a su vez se dividen en varias capas que proporcionan las funcionalidades necesarias, tal y como se muestra en la Figura 41.

El bloque *Application* es la parte más alta de la pila y la responsable de contener la lógica, la interfaz de usuario, así como la gestión de datos relacionados con el caso de uso actual que implementa la aplicación.

La capa *Host* es la parte intermedia de la pila de protocolos y representa el software que gestiona la comunicación entre dos o más dispositivos. Esta parte de la pila de protocolos contiene:

- Perfil de Acceso Genérico (GAP, *Generic Access Profile*).
- Perfil de Atributo Genérico (GATT, *Generic Attribute Profile*).
- Protocolo de Control y Adaptación de Enlace Lógico (L2CAP, *Logical Link Control and Adaptation Protocol*).
- Protocolo de Atributos (ATT, *Attribute protocol*).
- Administrador de Seguridad (SMP, *Security Manager Protocol*).
- Interfaz de Controlador de *Host* (HCI, *Host Controller Interface*), lado del *Host*.

El grupo *Controller* está directamente relacionado con el dispositivo físico, siendo esta capa la que permite transmitir y recibir señales de radio e interpretarlas como paquetes de información. Esta parte de la pila de protocolos contiene:

- Interfaz de Control de *Host* (HCI, *Host Controller Interface*), lado del *Controller*.
- Capa de Enlace (LL, *Link Layer*).
- Capa Física (PHY, *Physical Layer*).

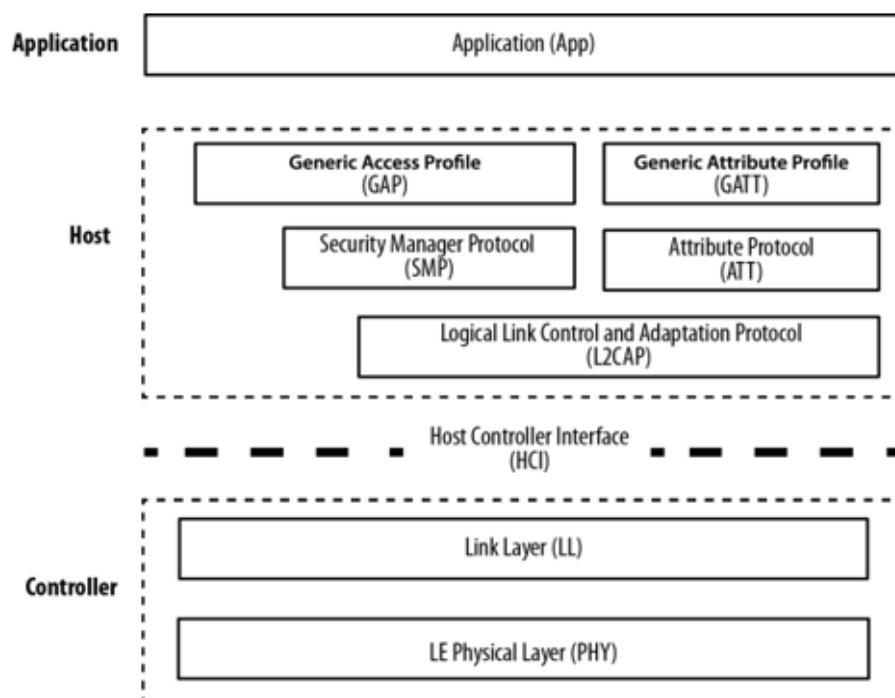


Figura 41. Pila de protocolos del estándar BLE.[28]

6.3.1 Capa Física

La capa física es la que se encarga de enviar las señales al aire, transmitiendo y recibiendo bits mediante ondas radio en la banda de frecuencia ISM. Existen 3 canales dedicados para el procedimiento de *Advertising*, y 37 para la transmisión de datos. Los canales 37, 38, y 39 son usados solo para el envío de paquetes de *Advertising*. El resto se utiliza para el intercambio de datos durante la conexión, tal y como se recoge en la Figura 42.

Debido a que coexisten otras tecnologías en el mismo espectro, los canales de *Advertising* se han situado estratégicamente para evitar interferencias. Además, en estado de conexión, BLE utiliza la técnica de *Frequency Hopping Spread Spectrum* (FHSS) para reducir interferencias.

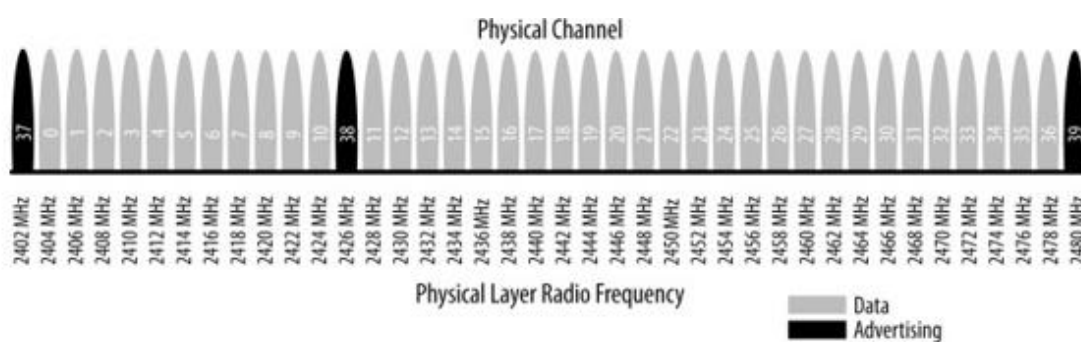


Figura 42. Canales de frecuencia [28].

6.3.2 Capa de enlace

Esta capa se encarga principalmente de los procesos de *Advertising* y *Scanning*, además, se responsabiliza del control, la negociación y el establecimiento de los enlaces, selecciona la frecuencia de transmisión de datos, y realiza el mantenimiento de las diversas formas de intercambio de datos y topologías. Su funcionamiento se describe mediante los siguientes estados:

- *Standby*: Estado por defecto de la capa de enlace al que se puede acceder desde cualquiera de los otros estados, y en el cual no se reciben ni se transmiten paquetes.
- *Scanning*: Los dispositivos de tipo *Central* utilizan este estado para escuchar los paquetes de *Advertising* enviados por el *Advertiser* a través de los canales designados. Se denomina *Advertiser* al dispositivo BLE que utiliza los canales de *Advertising* para anunciar que es conectable y detectable, o que puede ser descubierto. El dispositivo *Advertiser* puede solicitar información adicional con el objetivo de explorar los dispositivos BLE existentes. Los dispositivos que se encuentran dentro de este estado se denominan *Scanners*.

- *Advertising*: Los dispositivos de tipo *Peripheral* utilizan este estado para transmitir paquetes de *Advertising* en sus respectivos canales. Este tipo de dispositivo también utilizan este estado para atender las peticiones de los dispositivos de tipo *Central* cuando envían paquetes de *Advertising*.
- *Initiating*: Este es el estado en el que entra el dispositivo *Central*, estableciendo una conexión a modo de respuesta a los paquetes de *Advertising* recibidos, antes de pasar al estado *Connection*. Los dispositivos que se encuentran dentro de este estado se denominan *Initiators*.
- *Connection*: A este estado se entra cuando el dispositivo *Central* está estableciendo una conexión con otro dispositivo *Peripheral*. De esta forma se asignan los roles de *Master* y *Slave*. Cuando se ingresa desde el estado *Initiating* el dispositivo actúa como *Master*, mientras que cuando se entra desde el estado *Advertising* el dispositivo actúa como *Slave*. Los dispositivos *Master* pueden establecer múltiples conexiones con dispositivos *Slave*, y generalmente realizan búsquedas de otros dispositivos. Por otro lado, un dispositivo *Slave*, sólo puede tener una conexión con un único dispositivo *Master*. Los canales de datos se utilizan una vez establecida la conexión entre ambos dispositivos.

Todos estos estados pueden reagruparse en una máquina de estados para describir el funcionamiento de la capa de enlace, tal y como se muestra en la Figura 43.

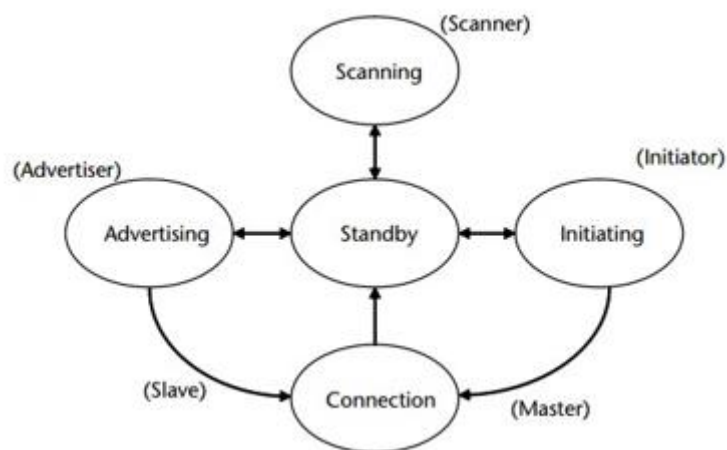


Figura 43. Máquina de estados de la capa de enlace.[28]

6.3.3 Interfaz de Control de *Host*

Esta interfaz sirve de pasarela entre las capas superior e inferior de la pila de protocolos. Las capas superiores generalmente residen en un *Host*, y las capas inferiores residen en un chip controlador de *Bluetooth* separado. La interfaz HCI proporciona un mecanismo de comunicación entre el *Host* y el Controlador *Bluetooth* vía comandos HCI.

6.3.4 Protocolo de Control y Adaptación de Enlace Lógico

El protocolo de control y adaptación de enlace toma múltiples protocolos de las capas superiores y los encapsula en el formato de paquete BLE estándar y viceversa. A la hora de transmitir este protocolo toma paquetes de elevado tamaño de las capas superiores y los divide en fragmentos, mientras que a la hora de recibir los recombina en un solo paquete, que luego se enviará en sentido ascendente a la entidad apropiada en las capas superiores del *Host*.

6.3.5 Protocolo de Atributos

El protocolo de atributos es un mecanismo de muy bajo nivel que básicamente define como se transmite una unidad de datos (un atributo). Un atributo está compuesto por tres elementos:

- un identificador de 16 bits.
- un UUID (Universally Unique Identifier) que define el tipo de atributo.
- un valor de una longitud determinada.

El valor es una matriz de bytes de cualquier tamaño. El significado real del valor depende completamente del UUID, y ATT no verifica si la longitud del valor es consistente con un UUID dado. El identificador es solo un número que identifica de forma única un atributo, ya que puede haber muchos atributos con el mismo UUID dentro de un dispositivo. ATT en sí no define ningún UUID, de esto se encarga el perfil de atributo genérico y perfiles de nivel superior.

Un servidor ATT almacena atributos. Un cliente ATT no almacena nada; utiliza el protocolo de conexión ATT para leer y escribir valores en los atributos del servidor. Puede haber permisos de seguridad asociados con cada atributo, los cuales se almacenan dentro del valor y se definen mediante perfiles de nivel superior. El protocolo ATT no los conoce y no intenta interpretar los valores de los atributos para probar los permisos, ya que de este problema se encarga el GATT.

Este protocolo tiene algunas características interesantes, como buscar atributos por UUID u obtener todos los atributos dado un rango de control, por lo que el cliente no necesita saber los números de control de antemano, ni los perfiles de nivel superior tienen que codificarlos.

La mayor parte del protocolo ATT es cliente-servidor puro: el cliente toma la iniciativa, y el servidor responde. Pero ATT tiene capacidades de notificación e indicación, en las que el servidor toma la iniciativa de notificar a un cliente que un valor de atributo ha cambiado, lo que evita que el cliente tenga que sondear el atributo.

6.3.6 Administrador de seguridad

El administrador de seguridad es un protocolo que contiene una serie de algoritmos de seguridad diseñados para proporcionar la capacidad de generar e intercambiar claves de seguridad a la pila de protocolos *Bluetooth*, de manera que permitan a los dispositivos comunicarse de forma segura a través de un enlace cifrado para confiar en la identidad del dispositivo remoto, y finalmente, para ocultar la dirección pública de *Bluetooth* si es necesario, con el fin de evitar que dispositivos malintencionados rastreen un dispositivo específico.

6.3.7 Perfil de atributo genérico

Los perfiles definen esencialmente cómo deben usarse los protocolos para lograr un objetivo particular, ya sea genérico o específico. El perfil de atributo genérico puede considerarse la columna vertebral de la transferencia de datos BLE, ya que en él se define cómo se organizan y se intercambian los datos entre las aplicaciones. Se basa en el protocolo ATT para intercambiar datos entre dispositivos, e incorpora una jerarquía y un modelo de abstracción de datos en la parte superior. Mantiene la misma arquitectura cliente/servidor presente en ATT, pero ahora los datos están encapsulados en servicios, que constan de una o más características. Se puede considerar que cada característica es la unión de una parte de datos del usuario, junto con otra parte de información descriptiva sobre ese valor, como sus propiedades, el nombre visible para el usuario, y las unidades, entre otros. Este tipo de información descriptiva también es conocido como metadatos.

Roles GATT

Este perfil define los dos tipos de roles que pueden adoptar los dispositivos BLE:

- *Cliente*: Envía solicitudes a un *Servidor* y recibe respuestas y actualizaciones iniciadas por el *Servidor*. El cliente GATT inicialmente no tiene conocimiento sobre los atributos del *Servidor*, por lo que primero debe consultar acerca de la presencia y la naturaleza de estos. Después de completar el descubrimiento de servicios, puede comenzar a leer y escribir los atributos encontrados en el *Servidor*, así como a recibir actualizaciones iniciadas por el *Servidor*.
- *Servidor*: Recibe solicitudes de un *Cliente* y devuelve respuestas. También envía actualizaciones iniciadas por el *Servidor* cuando está configurado para hacerlo, y es el responsable de almacenar y poner a disposición del *Cliente* los datos del usuario, organizados en atributos.

UUIDs

Un UUID es un número de 16 bytes globalmente único. Los UUIDs se utilizan en muchos protocolos y aplicaciones distintas a *Bluetooth*. BLE emplea estos UUIDs para, entre otras funciones, identificar servicios y características. Para una mayor eficiencia, y dado que 16 bytes tomarían una gran parte de la longitud de carga de datos de los 27 bytes de la capa de enlace, la especificación BLE agrega dos formatos de UUID adicionales: UUID de 16 y 32 bits. Estos formatos abreviados sólo se pueden usar con los UUIDs definidos en la especificación *Bluetooth*. SIG proporciona UUIDs abreviados para todos los tipos, servicios y perfiles que define y especifica.

Atributos

Los atributos son la entidad de datos más pequeña definida por GATT y ATT. Son piezas direccionables de información que pueden contener datos de usuarios. Tanto GATT como ATT únicamente pueden operar con atributos, por lo que para que los clientes y servidores interactúen toda la información debe organizarse de esta manera. Cada atributo contiene información sobre el atributo en sí y sobre los datos actuales, en los campos que se describen a continuación:

- *Identificador*: Es la parte de cada atributo que hace que sea direccionable, y se garantiza que no cambiará entre transacciones o a través de las conexiones. El *Cliente* debe usar la función de descubrimiento para obtener los identificadores de los atributos que le interesan.

- *Tipo*: Se corresponde con el tipo del atributo, que no es más que un UUID. Determina el tipo de datos presentes en el valor del atributo, y hay mecanismos disponibles para descubrir atributos basados exclusivamente en su tipo.
- *Permisos*: Especifican qué operaciones de ATT se pueden ejecutar en cada atributo y con qué requisitos de seguridad específicos. ATT y GATT definen los siguientes permisos:
 - *Permisos de acceso*: Determinan si el *Cliente* puede leer o escribir (o ambos) un valor de atributo. Cada atributo puede tener uno de los siguientes permisos de acceso:
 - *None*: El atributo no puede ser leído ni escrito por un cliente.
 - *Readable*: El atributo puede ser leído por un cliente.
 - *Writable*: El atributo puede ser escrito por un cliente.
 - *Readable and Writable*: El atributo puede ser leído y escrito por el cliente.
 - *Cifrado*: Determina si se requiere de un cierto nivel de cifrado para que el cliente pueda acceder al atributo. Estos son los permisos de cifrado definidos por GATT:
 - *No encryption required*: El atributo es accesible en una conexión de texto sin cifrar.
 - *Unauthenticated encryption required*: La conexión debe cifrarse para acceder al atributo, pero las claves de cifrado no necesitan ser autenticadas.
 - *Authenticated encryption required*: La conexión debe cifrarse con una clave autenticada para acceder al atributo.
 - *Autorización*: Determina si se requiere permiso del usuario para acceder al atributo. Un atributo puede elegir entre requerir o no requerir de autorización:
 - *No authorization required*: El acceso al atributo no requiere de autorización.
 - *Authorization required*: El acceso al atributo requiere de autorización.

Todos los permisos son independientes entre sí, y pueden combinarse libremente por parte del servidor, que los almacena por atributo.

- *Valor*: Contiene los datos actuales asociados al atributo. No hay restricciones sobre el tipo de datos que puede contener, aunque su longitud máxima está limitada a 512 bytes, de acuerdo a la especificación *Bluetooth*. Esta es la parte de un atributo a la que un *Cliente* puede acceder para leer y escribir. Todas las demás entidades conforman la estructura del atributo, y el *Cliente* no puede modificarlas ni acceder a ellas directamente.

Jerarquía de Datos y Atributos

ATT se basa en todos los conceptos expuestos anteriormente para proporcionar una serie de paquetes, que permiten a un cliente acceder a los atributos en un Servidor. GATT va más allá al permitiendo el acceso y la recuperación de información entre el *Cliente* y el *Servidor*, siguiendo un conjunto conciso de reglas que constituyen el marco utilizado por todos los perfiles basados en GATT. La Figura 44 muestra la jerarquía de datos y atributos introducida por GATT.

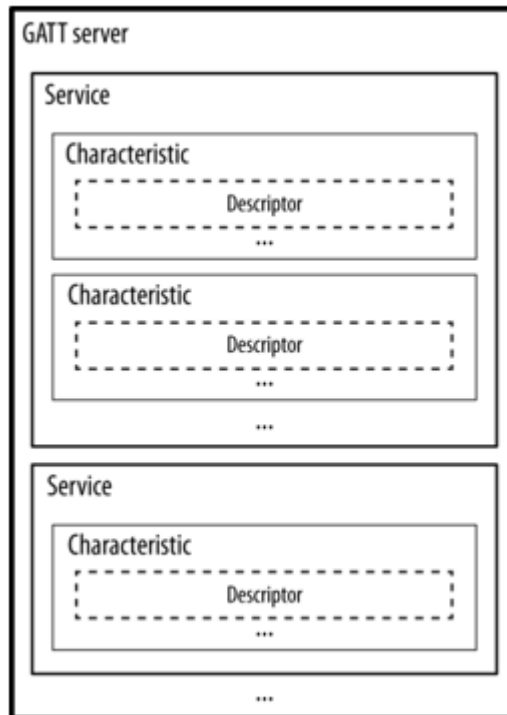


Figura 44. Jerarquía de datos y atributos[28]

En un Servidor GATT, los atributos se agrupan en servicios, cada uno de los cuales puede contener, o no, características. Estas características, a su vez, pueden incluir descriptors. Esta jerarquía se aplica estrictamente a cualquier dispositivo BLE, lo que significa que todos los atributos en un Servidor GATT están incluidos en una de estas tres categorías, sin excepción.

Servicios

Todos los atributos dentro de un servicio se conocen como la definición del servicio. Por lo tanto, los atributos de un servidor GATT se corresponden con una sucesión de definiciones de servicios, cada una comenzando con un único atributo que establece el comienzo de un servicio, denominado declaración del servicio, como se muestra en la Figura 45.

Handle	Type	Permissions	Value	Value length
0xNNNN	UUID _{primary service} or UUID _{secondary service}	Read Only	Service UUID	2, 4, or 16 bytes

Figura 45. Declaración del servicio [28]

El UUID del servicio primario, así como el del servicio secundario, son UUIDs estándar asignados por SIG que se usan como un tipo exclusivo para introducir un servicio, siendo su valor 0x2800 y 0x2801, respectivamente. Un Servicio Primario es el tipo estándar de servicio de GATT, que incluye la funcionalidad estándar relevante expuesta para el servidor GATT. Un servicio secundario, por contra, está incluido en los servicios primarios y tiene sentido sólo como su modificador, sin tener ningún significado real por sí mismo. En la práctica, los servicios secundarios rara vez se utilizan.

Características

Las características se pueden considerar como contenedores para los datos de usuario. Siempre incluyen al menos dos atributos: la declaración de característica, que proporciona metadatos sobre los datos de usuario actuales, y el valor de característica, que es un atributo completo que contiene los datos de usuario en su campo de valor. Además, el valor de característica puede ir seguido de descriptores, que amplían aún más los metadatos contenidos en la declaración de característica. La declaración, el valor y los descriptores forman la definición de característica, que es el conjunto de atributos que conforman una característica específica. La Figura 46 muestra la estructura de los primeros dos atributos de cada característica individual.

Handle	Type	Permissions	Value	Value length
0xNNNN	UUID _{characteristic}	Read only	Properties, value handle (0xMMMM), characteristic UUID	5, 7, or 19 bytes
0xMMMM	Characteristic UUID	Any	Actual value	Variable

Figura 46. Declaración y valor de una característica.[28]

El tipo del atributo de declaración de característica es un UUID único y estandarizado cuyo valor es 0x2803, y que se utiliza exclusivamente para indicar el comienzo de las características. Este atributo tiene permisos de solo lectura, ya que los clientes pueden recuperar su valor, pero en ningún caso modificarlo. La Figura 47 enumera los diferentes elementos concatenados dentro del valor del atributo de la declaración de característica.

Name	Length in bytes	Description
Characteristic Properties	1	A bitfield listing the permitted operations on this characteristic
Characteristic Value Handle	2	The handle of the attribute containing the characteristic value
Characteristic UUID	2, 4, or 16	The UUID for this particular characteristic

Figura 47. Valor de la declaración de características.[28]

El campo *Characteristic Value Handle* representa el identificador del atributo que contiene el valor actual de la característica, el campo *Characteristic UUID* representa el UUID de la característica, y el campo *Characteristic Properties* indica las operaciones y procedimientos que pueden realizarse con la característica. En la Figura 48 se muestran las propiedades de las características definidas por SIG.

Property	Location	Description
Broadcast	Properties	If set, allows this characteristic value to be placed in advertising packets, using the Service Data AD Type (see "GATT Attribute Data in Advertising Packets")
Read	Properties	If set, allows clients to read this characteristic using any of the ATT read operations listed in "ATT operations"
Write without response	Properties	If set, allows clients to use the Write Command ATT operation on this characteristic (see "ATT operations")
Write	Properties	If set, allows clients to use the Write Request/Response ATT operation on this characteristic (see "ATT operations")
Notify	Properties	If set, allows the server to use the Handle Value Notification ATT operation on this characteristic (see "ATT operations")
Indicate	Properties	If set, allows the server to use the Handle Value Indication/Confirmation ATT operation on this characteristic (see "ATT operations")
Signed Write Command	Properties	If set, allows clients to use the Signed Write Command ATT operation on this characteristic (see "ATT operations")
Queued Write	Extended Properties	If set, allows clients to use the Queued Writes ATT operations on this characteristic (see "ATT operations")
Writable Auxiliaries	Extended Properties	If set, a client can write to the descriptor described in "Characteristic User Description Descriptor"

Figura 48. Propiedades de una característica.[28]

El dispositivo cliente puede leer estas propiedades para determinar qué operaciones pueden realizarse sobre una característica, lo cual es especialmente importante para las propiedades *Notify* e *Indicate*, ya que estas operaciones son iniciadas por el dispositivo *Servidor*, pero requieren de su habilitación por parte del dispositivo *Cliente*, mediante el descriptor *Client Characteristic Configuration Descriptor*.

Un *Cliente* puede subscribirse para ser notificado cuando se modifique el valor de una característica, de manera que cuando se produce un cambio, el *Servidor* se lo notifica al *Cliente* mediante el

envío del nuevo valor. La diferencia entre *Indicate* y *Notify* es que en la primera de ellas el *Cliente* debe confirmar la recepción.

Por último, el atributo *Characteristic Value* de una característica contiene los datos que el *Cliente* puede leer o escribir para el intercambio de información. El tipo de este atributo es siempre el mismo UUID especificado en el campo *Characteristic UUID* de la declaración de característica.

Descriptores

Los descriptores se utilizan principalmente para proporcionar al *Cliente* información adicional sobre una característica y su valor. Siempre se especifican dentro de la definición de característica y después del atributo *Characteristic Value*. Los descriptores siempre están compuestos por un solo atributo, la declaración de descriptor de característica, cuyo UUID es siempre el tipo de descriptor, y cuyo valor contiene todo lo que define ese tipo de descriptor particular.

Uno de los descriptores más importantes y de uso común definidos por el GATT es el *Client Characteristic Configuration Descriptor*. Este descriptor es esencial para el funcionamiento de la mayoría de los perfiles y casos de uso. Su función es simple: actúa como un interruptor, habilitando o deshabilitando las actualizaciones iniciadas por el servidor, pero sólo para la característica a la que se encuentra asociado. Su valor no es más que un campo de dos bits, con un bit correspondiente a las notificaciones, y el otro a las indicaciones.

6.3.8 Perfil de Acceso Genérico

En el perfil de acceso genérico, existen cuatro roles que un dispositivo BLE puede adoptar para unirse a una red:

- *Broadcaster*: Optimizado para aplicaciones de solo transmisión, que distribuyen datos regularmente. Se envían periódicamente paquetes de *Advertising* con datos, y los datos están accesibles para cualquier dispositivo que esté escuchando. El rol *Broadcaster* utiliza el rol de *Advertiser* de la capa de enlace.
- *Observer*: Se utiliza en aplicaciones de solo recepción que desean recopilar datos de dispositivos de *Broadcasting*. Escucha los datos incluidos en los paquetes de *Advertising* de los dispositivos de *Broadcasting*. El rol de *Observer* utiliza el rol *Scanner* de la capa de enlace.
- *Central*: El rol *Central* corresponde al *Master* de la capa de enlace, al tratarse de un dispositivo capaz de establecer múltiples conexiones con otros dispositivos. El rol *Central*

es siempre el iniciador de las conexiones, y esencialmente permite que los dispositivos entren en la red

- *Peripheral*: El rol *Peripheral* corresponde al *Slave* de la Capa de Enlace, puesto que emplea paquetes de *Advertising* para permitir que los dispositivos *Central* lo encuentren y, posteriormente, establecer una conexión con él.

Cada dispositivo particular puede operar en uno o más roles a la vez, y la especificación BLE no impone restricciones al respecto. Los roles *Cliente/Servidor* GATT dependen exclusivamente de la dirección en la que fluyen las solicitudes de datos y las transacciones de respuestas, mientras que los roles GAP se mantienen constantes, no existiendo ninguna relación entre estos dos roles.

6.4 Dispositivo BLE final

El dispositivo BLE elegido para realizar la integración con el asistente Virtual Alexa es *PLAYBULB Candle* de Mipow, representado en la Figura 49 [29]. Este dispositivo simula la luz de una vela con un LED multicolor, con el que es capaz de reproducir infinitas combinaciones desde una aplicación móvil *PLAYBULB X*. La comunicación entre el dispositivo y la aplicación móvil se realiza vía BLE.



Figura 49. *PLAYBULB Candle*.

El objetivo principal de este TFG es conseguir realizar la integración de este dispositivo con el asistente virtual Alexa, dada la incompatibilidad inicial de Alexa para comunicarse con dispositivos que gestionan sus comunicaciones mediante el protocolo BLE, utilizando el dispositivo *RedBear DUO* como pasarela entre ambos para tratar de reproducir mediante comandos de voz, algunas de las funcionalidades que ofrece la aplicación móvil del dispositivo *PLAYBULB Candle*. Una vez recibido el comando de voz, Alexa se comunicará con el dispositivo *DUO* para que éste

desempeñe el rol de dispositivo *Central*, reproduciendo las funcionalidades de encender/apagar el LED o cambiar el color actual del LED por rojo, verde o azul de la aplicación móvil proporcionado con el dispositivo *PLAYBULB Candle*.

6.5 Alexa Echo Dot

El dispositivo *Echo Dot* es un altavoz con micrófonos incorporados que están escuchando constantemente, esperando a que el usuario diga “Alexa” seguido de una pregunta, una orden o algún comando específico. La Figura 50 representa la apariencia de este dispositivo.



Figura 50. Alexa Echo Dot

Echo Dot es el dispositivo más básico para interactuar con Alexa. Utiliza un reconocimiento de voz de campo lejano, es decir, aunque haya ruido ambiente será capaz de reconocer la voz del usuario. La calidad del sonido que emite es su punto débil, pero se puede conectar por *Bluetooth* o mediante un cable a otros equipos de sonido. Tiene forma curvada, un acabado de tela, y está disponible en color antracita, gris claro o gris oscuro. Para que el *Echo Dot* pueda interactuar con Alexa, el usuario debe asegurarse de conectarlo a una red Wi-Fi. Para ello, basta con descargarse la aplicación *Amazon Alexa* en el móvil, seleccionar el dispositivo de Amazon a configurar, y seguir los pasos indicados en la aplicación.

Este dispositivo permite preguntar por el tiempo que hará, escuchar las noticias, e incluso encender las luces si está conectado a un sistema de domótica. Además, se pueden añadir a la cesta de compras de Amazon cualquier producto que se le pida

Por otra parte, los desarrolladores pueden usar el *Alexa Skills Kit* para dar a Alexa nuevas funcionalidades, como por ejemplo solicitar horarios de trenes, escuchar una cadena de radio en concreto, pedir información de un vuelo o solicitar algo de comida a domicilio.

Alexa Voice Service ayuda a los desarrolladores a incluir Alexa en los productos. Compañías como Sonos, Bose, Energy Sistem, Grundig, Huawei, Sony o Hama, entre otras, podrán incorporar en los equipos este asistente de voz. La utilización de Spotify o Amazon Prime Music están integradas con Alexa.

6.6 Diagrama de flujo

En la Figura 51 se representa un diagrama de flujo en el que se refleja el proceso que se llevará a cabo en la plataforma desarrollada en este TFG desde el momento en el que se le indica a Alexa el comando, hasta que se enciende el LED del dispositivo *PLAYBULB Candle*.

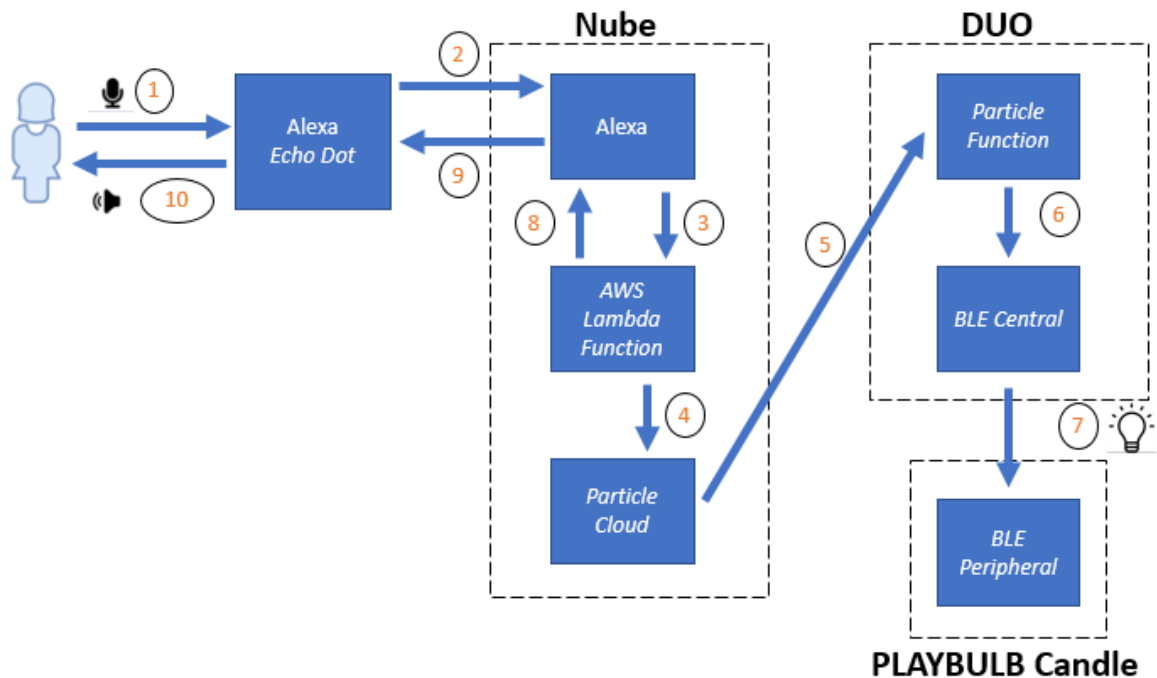


Figura 51. Diagrama de flujo Alexa, DUO y PLAYBULB Candle.

A continuación, se describen las funcionalidades de los procesos que aparecen en el diagrama de flujo:

1. El usuario le comunica a Alexa el comando por medio de voz.
2. El dispositivo *Echo Dot* recibe el comando y lo transfiere a los servidores de Alexa en la nube.
3. Alexa analiza el comando recibido e inicia la función *AWS Lambda* correspondiente.
4. La función contesta con una respuesta que enviará posteriormente a Alexa (funcionalidad 8), para lo cual deberá iniciar con antelación la comunicación con *Particle Cloud*.
5. *Particle Cloud* comienza la comunicación con el dispositivo *DUO* mediante las *Particle function*.
6. *Particle function* activa la funcionalidad que el usuario pretendía ejecutar, bien sea encender/apagar el LED como cambiar el color.
7. El dispositivo *DUO*, con el rol de *Central* inicia la transmisión de paquetes BLE al dispositivo *PLAYBULB Candle*, y éste ejecuta la funcionalidad asociada con el comando recibido.

8. La función *AWS Lambda* contesta al comando recibido en la funcionalidad 3, enviando una respuesta al servidor de Alexa.
9. Alexa prepara la respuesta recibida y se la envía al dispositivo *Amazon Echo Dot*, para que se la transmita al usuario.
10. El dispositivo *Amazon Echo Dot* recibe la respuesta de los servidores de Alexa y se la comunica al usuario.

Es importante recalcar que la funcionalidad 7 se lleva a cabo porque el dispositivo *DUO* ha establecido una conexión previa con el dispositivo final *PLAYBULB Candle*. El dispositivo *DUO* intenta establecer esta conexión desde el momento en el que se alimenta, tal y como se refleja en el diagrama de flujo asociado a la funcionalidad del dispositivo *DUO* como *BLE Central*, mostrado en la Figura 52

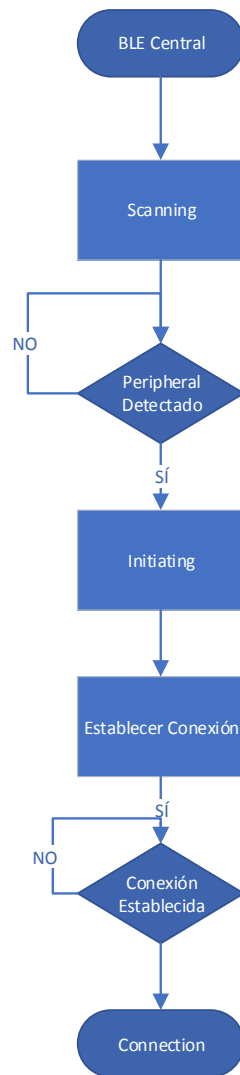


Figura 52. Diagrama de flujo BLE Central

7. RedBear DUO

7.1 Descripción y características

DUO es una placa de desarrollo creada por la empresa *RedBear*, para simplificar el proceso de creación de productos de Internet de las Cosas. Integra un microcontrolador MCU Cortex-M3, un módulo inalámbrico que proporciona conectividad Wi-Fi y BLE, así como una gran cantidad de periféricos. *DUO* permite desarrollar aplicaciones para comunicarse con otros accesorios inalámbricos o sus dispositivos inteligentes, por ejemplo: un teléfono móvil, una tablet o un ordenador, a través de BLE, Wi-Fi o ambos al mismo tiempo. Si el dispositivo *DUO* se conecta a un *router* que se encuentre conectado a la red, incluso puede comunicarse con el *DUO* a través de la nube [30].

DUO es compatible con varios tipos de lenguaje de programación, como *Arduino*, *C/C ++*, *JavaScript* y *Python*. Además, se puede desarrollar las aplicaciones para *DUO* utilizando *GCC*, *Arduino IDE*, *Particle Web IDE*, *Espruino Web IDE* y *Broadcom WICED SDK*, aunque en este TFG en particular solo se utilizará *Particle Web IDE*.

Las principales características de este dispositivo son las siguientes [31]:

- Chip STM32F205 de STMicroelectronics, con microcontrolador ARM 32 bits Cortex-M3 que funciona a 120 MHz
- Módulo Wi-Fi inalámbrico AMPAK AP6212A (construido en torno al chip Broadcom BCM43438):
 - Wi-Fi 802.11b / g / n, que trabaja en la Banda ISM de 2.4 GHz
 - *Bluetooth 4.1* (modo dual), trabajando en la banda ISM de 2.4 GHz
- Memoria:
 - *Flash* interna de 1 MB.
 - *Flash SPI* externa de 2 MB.
 - 128KB SRAM.
- Capacidades de E/S:
 - 18 x E/S digital.
 - 8 x entrada ADC.
 - 2 x salida DAC.
 - 13 x PWM.
- Conectividad:
 - 2 x UART.

- 2 x SPI.
 - 1 x I2S.
 - 1 x I2C.
 - 1 x CAN.
 - 1 x USB de alta velocidad.
 - Puerto de depuración JTAG (SWD).
- Sistema operativo en tiempo real.
 - Hardware y software código abierto.
 - Antena con chip de señal alternativa o antena externa.
 - Certificado FCC y CE.

7.2 Diagrama de bloques y pines de E/S

En la Figura 53 se muestra el diagrama de bloques del dispositivo *DUO*, mientras que en La Figura 54 desglosa un patillaje del DUO.

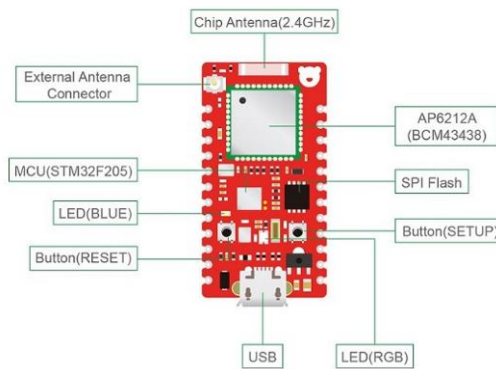


Figura 53. Diagrama de bloques del DUO.

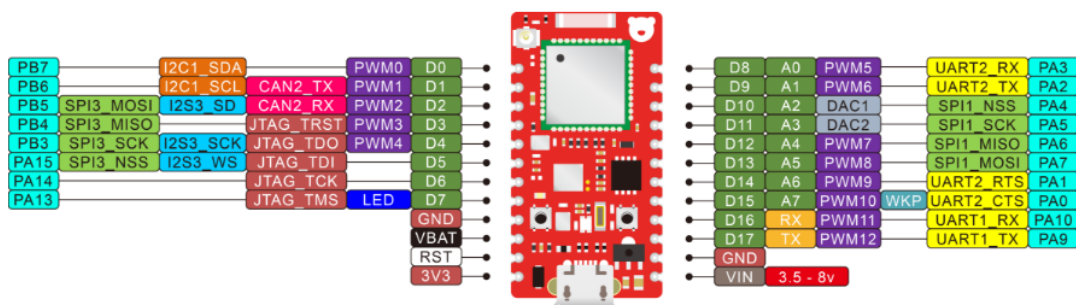


Figura 54. Patillaje del DUO

En cuanto a la fuente de alimentación principal, hay que tener en cuenta que se debe utilizar solo una fuente de alimentación como fuente de alimentación principal a la vez; de lo contrario, la placa se dañará.

Para hacer que el dispositivo *DUO* funcione con normalidad, la tensión de trabajo a nivel de la placa (VCC) debe oscilar entre 3.0V ~ 3.6V. Si el voltaje de funcionamiento es inferior a 3.0V, el módulo inalámbrico no funcionará bien, y si es superior a 3.6V, causará daños en la placa. Hay tres maneras de alimentar el dispositivo *DUO*

- Puerto micro USB:

Al conectar el dispositivo *DUO* a una fuente de alimentación de 3.5V~ 8V o a un ordenador a través de un cable micro USB, éste se encenderá. El regulador de la placa regula primero la tensión de entrada del puerto USB a 3.3 V y luego alimenta las unidades de circuito de la placa completa. En este caso, el pin *VIN* emitirá una tensión cercana al voltaje de entrada del puerto USB y el pin *3V3* proporcionará una tensión igual al voltaje del nivel de la placa, es decir, 3.3V. Entonces el pin *VIN* se puede utilizar para alimentar otros dispositivos, como por ejemplo relés, servidores, motores, etc. Además, el pin *3V3* también se puede utilizar para alimentar otros dispositivos con una tensión de 3.3V.

- PIN *VIN*:

Conectar el pin *VIN* a una fuente de alimentación de 3.5V ~ 8V hará que se encienda el dispositivo *DUO*. La tensión de entrada del pin *VIN* será regulada a 3.3V por el regulador de la placa primero y luego alimentará a todas las unidades del circuito de la placa. En este caso, el pin *3V3* suministrará una tensión igual al voltaje del nivel de la placa, es decir 3.3V.

- PIN *3V3*:

Conectando el pin *3V3* a una fuente de alimentación de 3.0V ~ 3.6V también se encenderá el *DUO*. El voltaje de este pin abastece directamente a las unidades de circuito de la placa completa.

- Fuente de alimentación de respaldo

El voltaje del pin *VBAT* debe oscilar entre 1.8V y 3.6V, pero no puede alimentar el *DUO*. Según el manual de referencia técnica del chip STM32F205, el pin *VBAT* se puede conectar a un voltaje de reserva opcional suministrado por una batería, o por otra fuente, para retener el contenido de los registros de respaldo RTC, hacer una copia de seguridad de SRAM, y alimentar el RTC cuando VDD esté apagado. Por lo tanto, está bien conectar el dispositivo *DUO* a una fuente de alimentación principal y a una fuente de alimentación de respaldo al mismo tiempo.

7.3 Conectando el DUO

Existen dos formas de conectar el dispositivo *DUO*, la primera mediante el uso de un dispositivo móvil, por lo cual, hay que acceder a la *App Store* de iOS y descargarse la aplicación, o bien descargarse la APK de Android en el *smartphone* [32].

Una vez que esté instalada hay que iniciar la aplicación. Lo primero que hará será verificar que el último *firmware* disponible en *GitHub* coincide con el *firmware* que contiene la aplicación para, en caso de que exista un nuevo *firmware* disponible, descargarlo en el dispositivo iOS / Android y posteriormente actualizar el dispositivo *DUO* a través de TCP / IP local. En caso de no tener acceso a la red, la aplicación mostrará un mensaje indicando que no se puede descargar la última versión sin conexión a Internet. En la Figura 55 se recoge lo expuesto anteriormente.

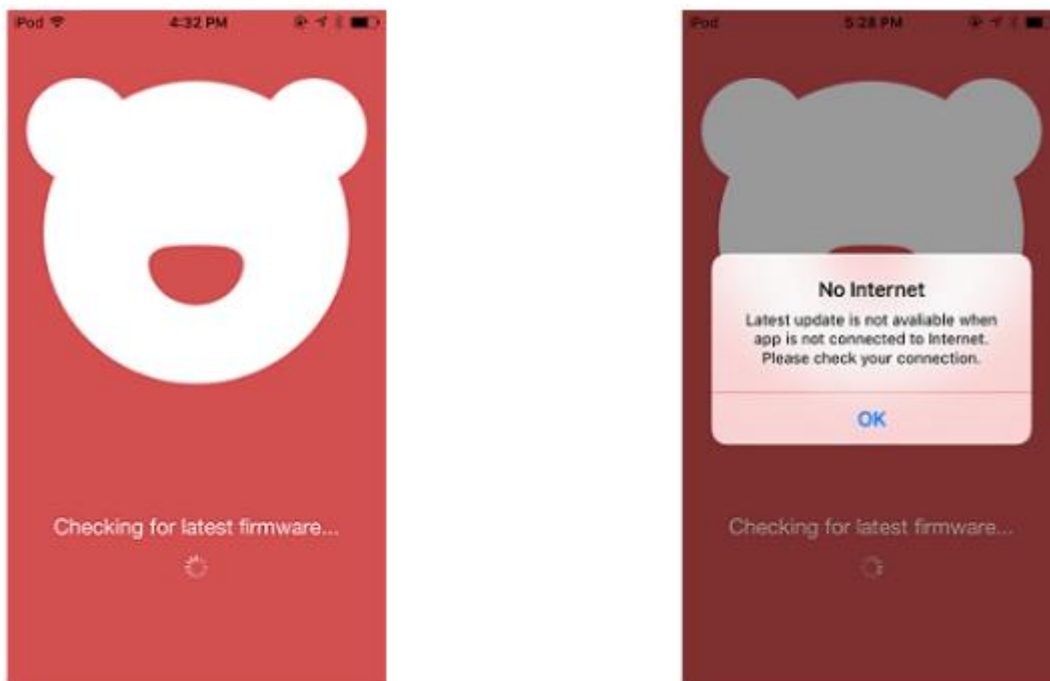


Figura 55. Iniciando RedBear Duo App (con internet y sin internet)

Tras comprobar la versión del *firmware*, la aplicación ofrece dos maneras de configurar el dispositivo *DUO*: a través de WiFi o BLE, por lo que ahora el *DUO* debe ingresar en modo de escucha. En este modo, el dispositivo *DUO* actúa como WiFi *SoftAP* y BLE *Peripheral*. Las diferencias entre estas dos formas son que al usar WiFi se puede actualizar el *firmware* del sistema en caso de que exista una versión actualizada disponible en su dispositivo iOS / Android, y la contraseña de la credencial que va a configurar se encriptará antes de ser enviada, mientras que utilizando BLE no se puede actualizar el *firmware* del dispositivo *DUO* y la contraseña de la credencial se enviará en texto sin formato. En caso de que la función WiFi o la función BLE estén desactivadas, o no estén disponibles en el dispositivo iOS / Android, la subsección

correspondiente quedará inhabilitada. La Figura 56 representa las dos maneras de configurar el DUO cuando las funciones WiFi y BLE están activadas y desactivadas, respectivamente.

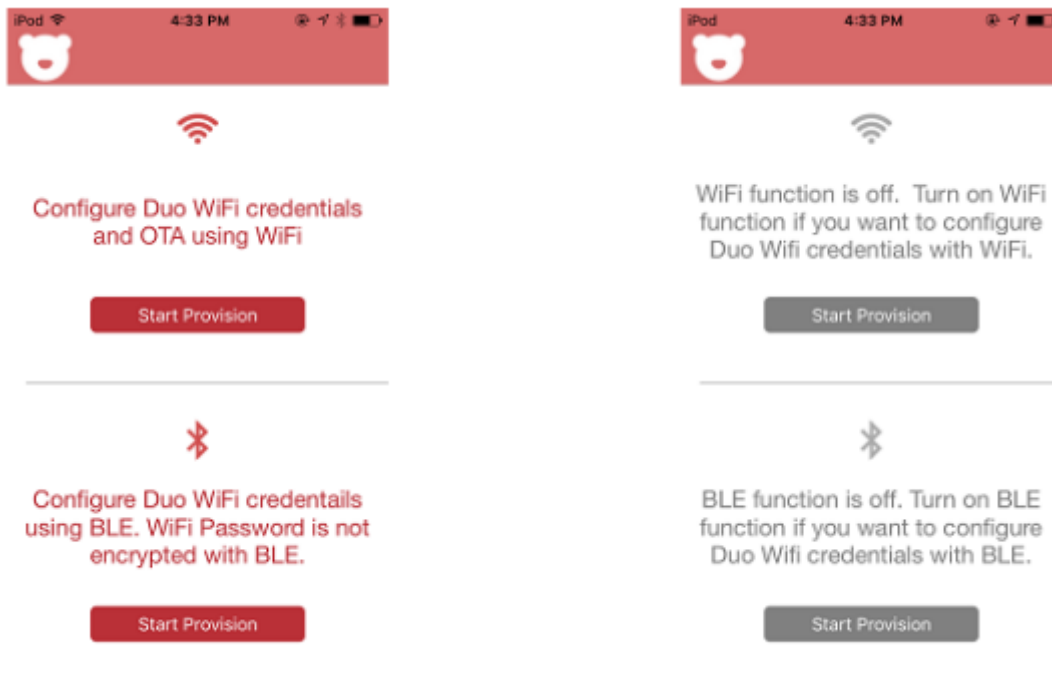


Figura 56. Configuración del DUO con las funciones WiFi y BLE activadas/desactivadas

Para configurar el dispositivo *DUO* a través de WiFi basta con seguir las instrucciones que se muestran en la Figura 57, o bien tocar el botón "Change WiFi Setting" para conectar el dispositivo iOS / Android al *SoftAP* emitido por el *DUO*. El SSID del softAP está en el formato "Duo-xxxx", donde el campo "xxxx" varía de un *DUO* a otro. Una vez que el dispositivo iOS / Android se haya conectado, hay que regresar a la aplicación.

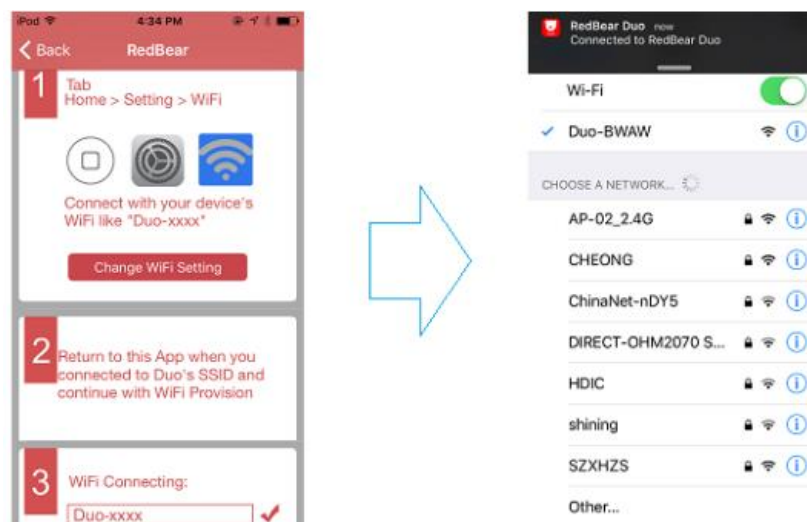


Figura 57. Instrucciones para configurar el DUO a través de WiFi

Por otro lado, para la configuración del dispositivo *DUO* a través de BLE, la aplicación mostrará una lista de todos los dúos en el dispositivo iOS / Android. Hay que seleccionar el *DUO* que corresponda en la lista de resultados del escaneo, tal y como se muestra en la Figura 58.

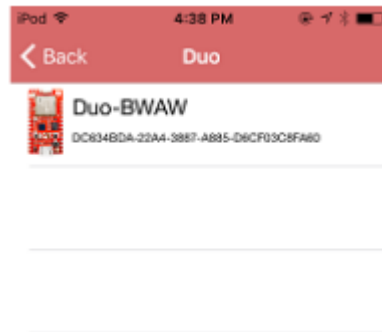


Figura 58. Configuración del *DUO* a través de BLE

En la Figura 59 se describe el siguiente paso a seguir una vez que el dispositivo iOS / Android esté conectado al dispositivo *DUO*, tal y como se puede observar automáticamente escaneará y listará las redes WiFi alrededor del *DUO*, siendo necesario seleccionar una de ellas para terminar de configurar el dispositivo *DUO*. También se puede desplegar la pantalla para escanear redes WiFi nuevamente.

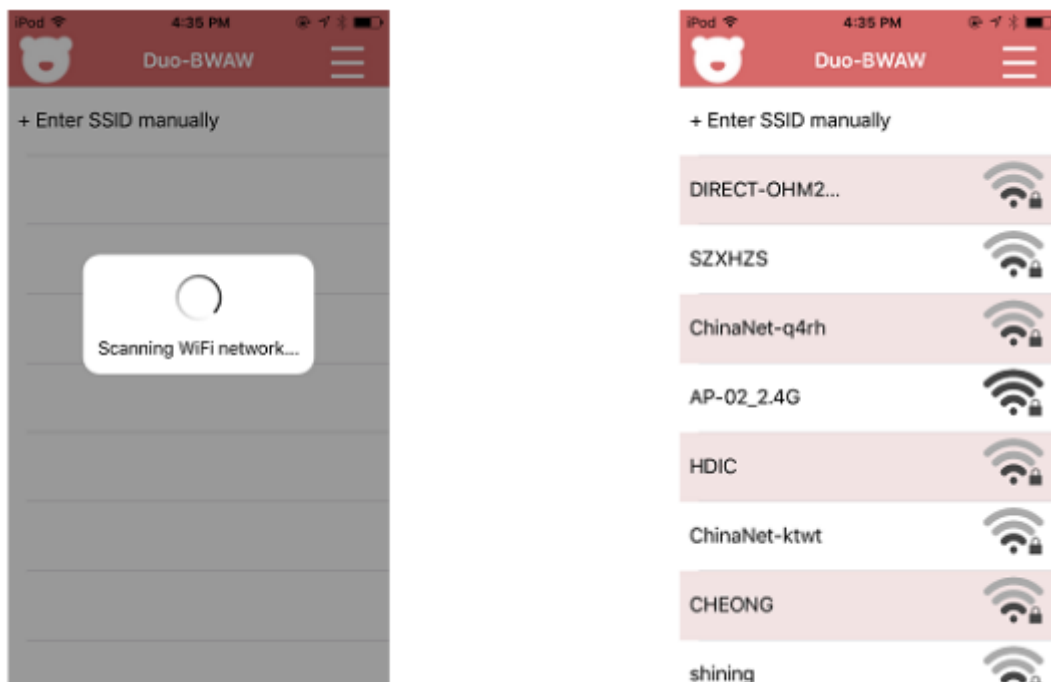


Figura 59. Configuración del *DUO* a través de una red WiFi cercana

La Figura 60 muestra cómo pulsando el botón de menú situado en la esquina superior derecha, se puede buscar la identificación del dispositivo *DUO* y la versión del *firmware* instalada.

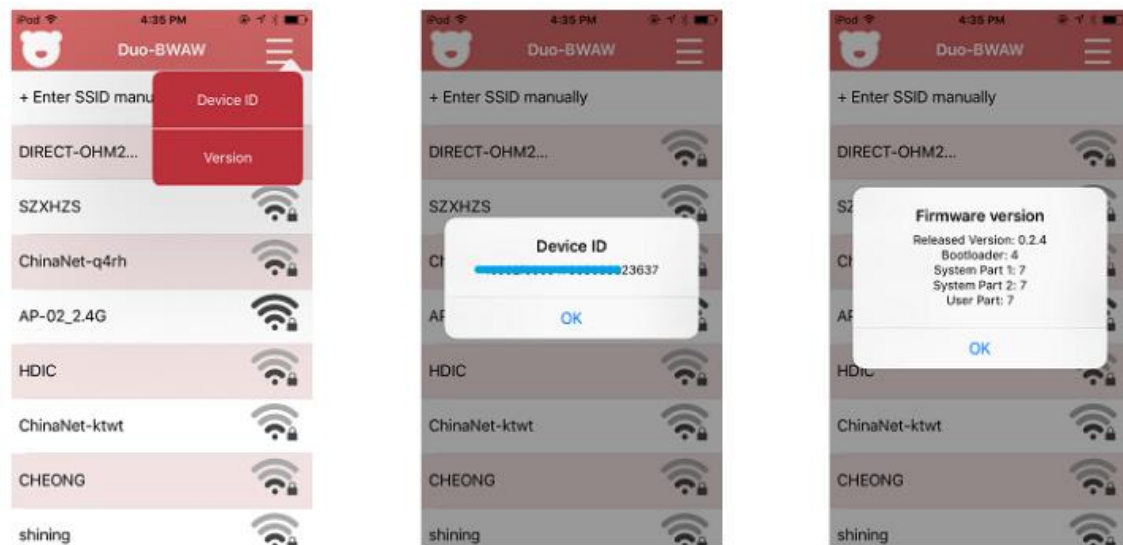


Figura 60. Identificación del *DUO* y versión del *firmware* actual

A continuación es necesario elegir una de las redes WiFi de la lista para la configuración del *DUO*. Si la red WiFi que desea utilizar no está en la lista, se puede configurar manualmente tocando en la primera columna "+ Enter SSID manually". Si la red WiFi elegida no es abierta, se deberá ingresar la contraseña correspondiente. Después de configurar la credencial WiFi, el dispositivo *DUO* desconectará la conexión WiFi / BLE del dispositivo iOS / Android, e intentará conectarse a la red WiFi seleccionada, con el RGB parpadeando en verde. Si el dispositivo *DUO* se conecta al AP con éxito, entonces se reiniciará. De lo contrario, *DUO* volverá a ingresar al modo de escucha para que vuelva a configurar las credenciales de Wi-Fi.

Si el dispositivo *DUO* está conectado utilizando WiFi, la aplicación salta a la ventana mostrada en la Figura 61, ya que no sabe si el dispositivo se ha conectado correctamente a la red WiFi o no, debido a la implementación del protocolo *SoftAp* que utiliza la aplicación actualmente.

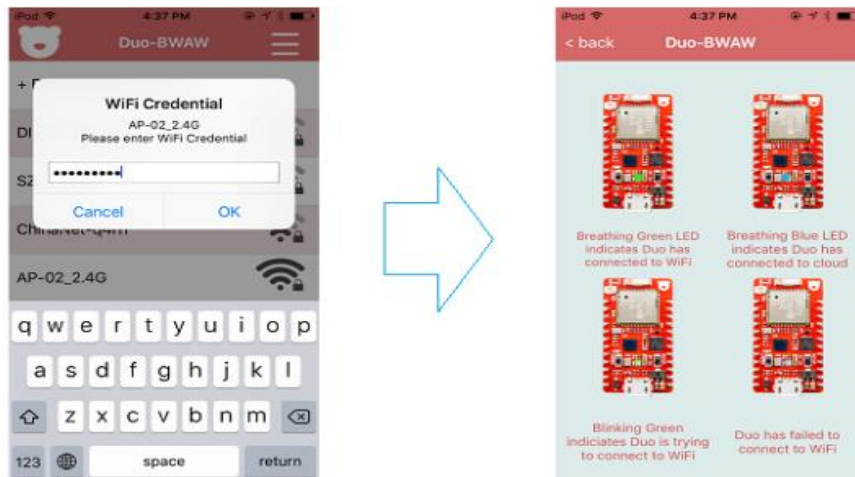


Figura 61. Conectando el DUO a una red WiFi

Si se conecta al *DUO* utilizando BLE, la aplicación espera hasta que el dispositivo se conecte a la red WiFi con éxito y luego muestra por pantalla la información de la IP. La Figura 62 muestra una descripción gráfica de lo comentado anteriormente. Si se produce un tiempo de espera debido a una contraseña incorrecta o alguna otra causa, la aplicación simplemente vuelve a mostrar la ventana de configuración del dispositivo *DUO* por medio de las dos opciones disponibles.

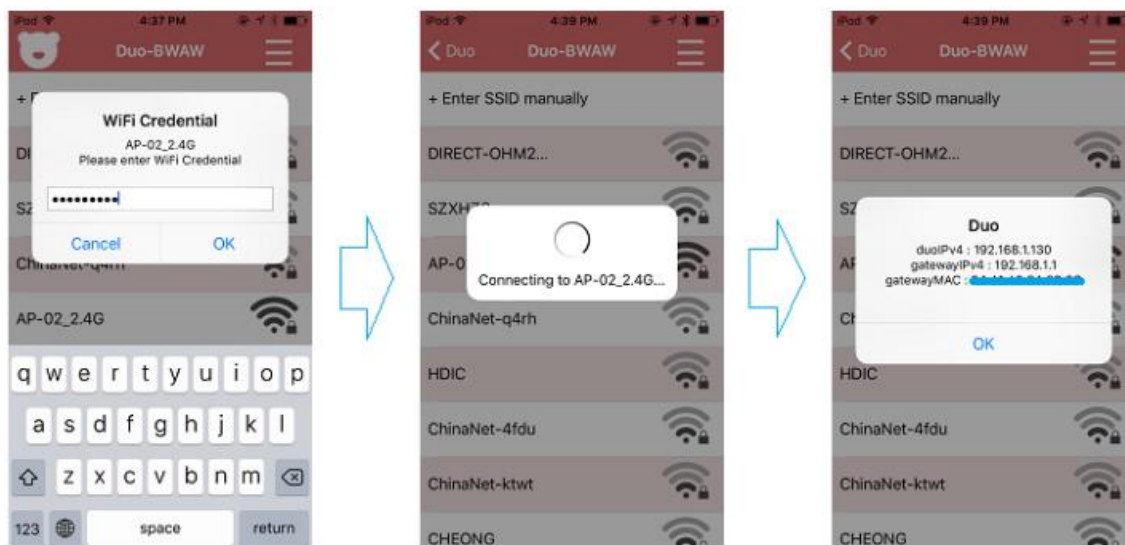


Figura 62. Conectando a una red WiFi un DUO enlazado por BLE

La otra forma de conectar el dispositivo *DUO* es mediante un terminal de puerto serie. El primer paso consiste en conectar el *DUO* al ordenador directamente a través de su puerto USB nativo con un cable micro USB. Al conectarlo, el dispositivo *DUO* entrará en modo de escucha permitiendo la configuración de las credenciales Wi-Fi. Existen dos maneras de continuar con el proceso de conectar el *DUO* mediante un terminal de puerto serie, dependiendo del tipo de sistema operativo que haya instalado en el ordenador que se va a utilizar.

- Windows: Instalar en el ordenador el driver del dispositivo *DUO* [33] y el terminal serie *PuTTY*.

- OSX y Linux: Iniciar la aplicación Terminal y usar el comando *screen*. En Linux, se puede instalar *screen* con el siguiente comando:

```
$ sudo apt-get install screen
```

Para ejecutar *screen* una vez instalado hay que utilizar este comando:

```
$ screen /dev/tty.ACMX, donde ACMX es el puerto serie del DUO
```

El comando en OSX para ejecutar *screen* es el siguiente:

```
$ screen /dev/tty.usbmodemXXXXX, donde XXXXX es el puerto serie del DUO.
```

Los pasos que se indican a continuación son iguales para todos los ordenadores, independientemente del tipo de sistema operativo instalado. De este modo, si se desea comprobar la versión actual del *firmware* del dispositivo *DUO*, basta con escribir “v” en el terminal; Con este procedimiento se puede comprobar si el dispositivo necesita llevar a cabo una actualización. Para localizar el *ID* del dispositivo habrá que ejecutar el comando “i” en el terminal. El *ID* se utilizará posteriormente para reclamar el *DUO* en el entorno Particle Cloud, por lo tanto, es importante guardar una copia. Para establecer las credenciales WiFi se deberá ejecutar el comando “w” en el terminal, solicitándole al usuario y contraseña para asociar el *DUO* con el punto de acceso.

Si todo ha ido bien, el dispositivo *DUO* estará en modo de escucha e intentará conectar con el AP mientras el RGB está parpadeando verde. En caso de que el *DUO* consiga conectarse se reiniciará. En caso contrario el *DUO* entrará de nuevo en modo de escucha esperando a que se reinicien las credenciales WiFi. Para asegurarse de que el dispositivo *DUO* se ha conectado correctamente hay que abrir un navegador y escribir como URL “*duo.local*” si el navegador soporta *mDNS*. En caso contrario, habrá que escribir <http://192.168.1.11>. En la Figura 63 se puede observar la aplicación que está ejecutándose en el *DUO*




Figura 63. Aplicación ejecutándose en el *DUO*

A continuación, seleccionando el botón "HIGH" debería encenderse el LED del dispositivo *DUO*, y pulsando el botón "LOW", apagarse.

7.4 Reclamar el DUO

Para poder utilizar las funciones de *Particle* es necesario reclamar el dispositivo en el entorno Particle Cloud.

Para ello, lo primero que hay que hacer es acceder al Particle Build IDE por medio de una cuenta o

bien registrándose si no se tiene una. Seleccionando la opción Devices  que se encuentra en la esquina inferior izquierda de la pantalla, se accede a una pestaña en la que se pueden visualizar los dispositivos que están vinculados con Particle Cloud. Además, se pueden añadir nuevos dispositivos pulsando sobre el botón "ADD NEW DEVICE", tal y como se muestra en la Figura 64.

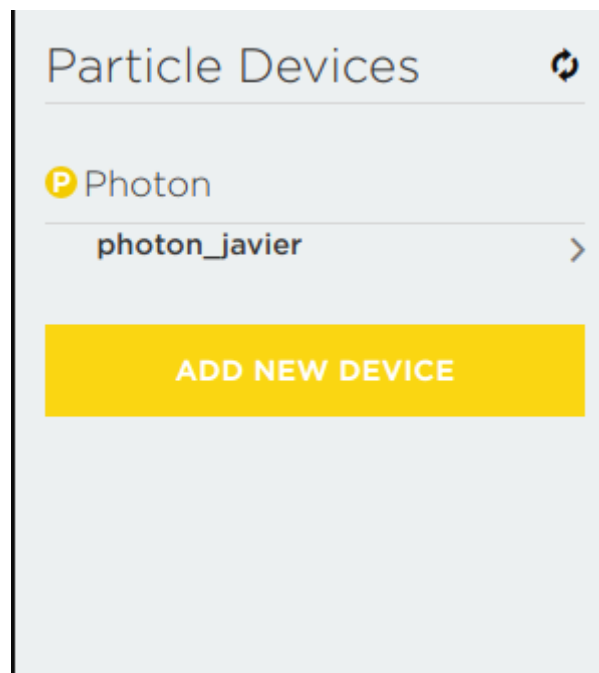


Figura 64. Dispositivos conectados a Particle Cloud

Una vez que se haya pulsado en este botón, habrá que introducir el *ID* del dispositivo *DUO*, guardado anteriormente, y presionar el botón "CLAIM A DEVICE" para reclamar el *DUO*, cuyo LED estará parpadeando en color celeste si se ha reclamado correctamente.

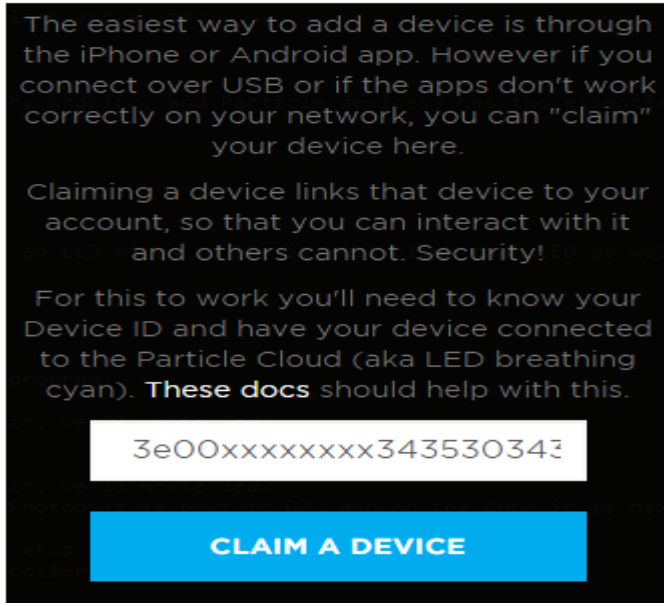


Figura 65. Reclamo del dispositivo

En ese caso, ya se podría asignar un nombre al dispositivo *DUO* y guardarlo, pulsando el botón "SAVE".

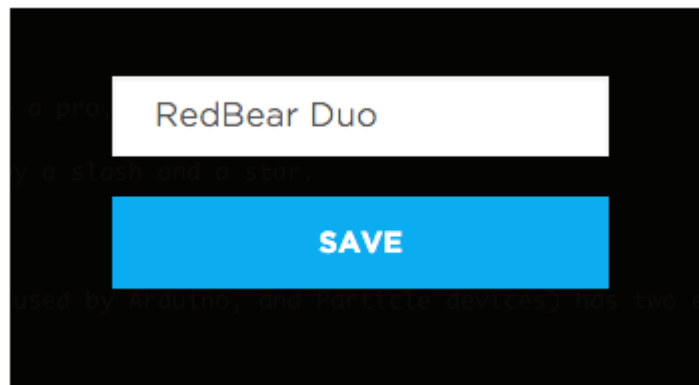


Figura 66. Renombrando el dispositivo

8. Integración de *PLAYBULB Candle* con Alexa mediante el dispositivo RedBear *DUO*

8.1 Descubrimiento de los valores del servidor GATT

Para que la comunicación entre el dispositivo *DUO*, actuando como BLE *Central*, con el dispositivo final *PLAYBULB Candle*, actuando como BLE *Peripheral*, sea posible, es necesario que el dispositivo *DUO* conozca una serie de valores que solo se encuentran disponibles en el servidor GATT, y que son imprescindibles para realizar una comunicación efectiva. Estos valores son los siguientes:

- El UUID de los servicios que se vayan a utilizar.
- El nombre del dispositivo.
- Identificar la característica que habilita el encendido/apagado y el cambio de color del LED en el dispositivo *PLAYBULB Candle*.
- Determinar las propiedades de la característica que se va a utilizar.
- Localizar el valor de los atributos que determinan el encendido, apagado e incluso el color del LED del dispositivo *PLAYBULB Candle*.
- Identificar la característica que permite el envío de notificaciones.

Para poder llevar a cabo el descubrimiento de todos estos valores se utilizó la herramienta *nRFConnect*. Esta herramienta es una aplicación móvil, disponible tanto para dispositivos Android como iOS, desarrollada por la empresa *Nordic Semiconductor*. La aplicación *nRFConnect* permite, entre otras funciones, al usuario escanear, enviar paquetes de *advertising*, y explorar los dispositivos BLE que se encuentren dentro del rango para comunicarse con ellos. La Figura 67 muestra el resultado obtenido tras realizar un escaneo con esta aplicación, en una versión Android, con el fin de conectarse al dispositivo *PLAYBULB Candle*.

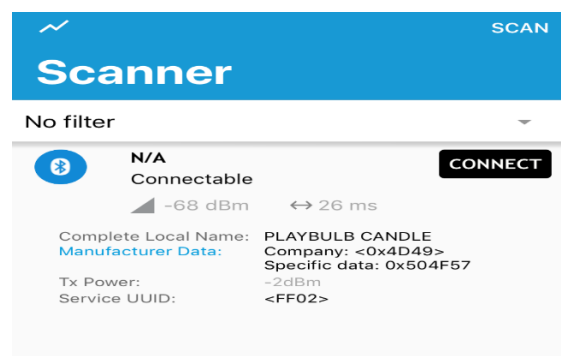


Figura 67. nRFConnect Scan

De este escaneo se puede extraer el nombre del dispositivo *Peripheral* al que se pretende conectar, que en este caso el nombre es *PLAYBULB CANDLE*. Al conectarse a este dispositivo

aparecen los servicios definidos, tal y como se muestra en la Figura 68, donde además se puede obtener el UUID de cada uno de ellos.

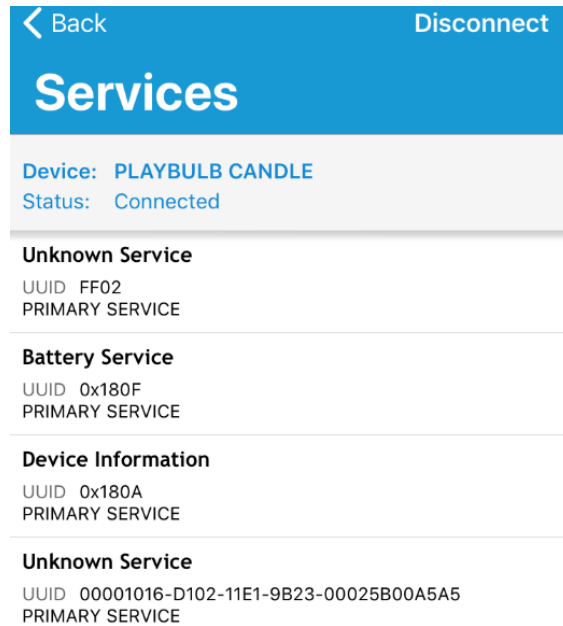


Figura 68. nRFConnect Servicios

Todos los UUID de los tres primeros servicios mostrados en la Figura 68 son UUID base, es decir todos tienen el formato 0x0000XXXX00001000800000805F9B34FB, donde **XXXX** representa el conjunto de bits que hace que el identificador sea único.

Accediendo a cada uno de los servicios se descubren las características definidas en cada uno. En este caso, se ha accedido al primer servicio denominado *Unknown Service*, con identificador FF02, tal y como se ve en la Figura 69. Para identificar las características que se utilizan para cambiar el estado del LED del dispositivo *PLAYBULB Candle*, basta con fijarse en aquellas cuyas propiedades permite lectura (*read*) y escritura (*write*). En este caso, las únicas características que permiten lectura y escritura son precisamente las características del servicio al que se acaba de acceder, por tanto, éste se denominará el servicio primario.

La única forma de identificar la característica que se encarga de cambiar el estado del LED es conociendo los valores del atributo que permiten estos cambios, es decir, una vez conocidos estos valores, habría que probar uno de ellos en cada una de las características, hasta dar con la que realice el cambio esperado en el estado del LED. Para poder descubrir los valores del atributo, se utilizar un dispositivo BLE *sniffer* con el fin de capturar y analizar los paquetes transmitidos en las diferentes comunicaciones BLE que se establezcan entre el dispositivo *PLAYBULB Candle* y la aplicación móvil *PLAYBULBX*, con el fin de determinar cada valor del atributo que consiguen los

estados del LED deseados. La aplicación *PLAYBULBX* es una aplicación desarrollada por la empresa *Mipow* que permite controlar el dispositivo *PLAYBULB Candle* con un *smartphone*.

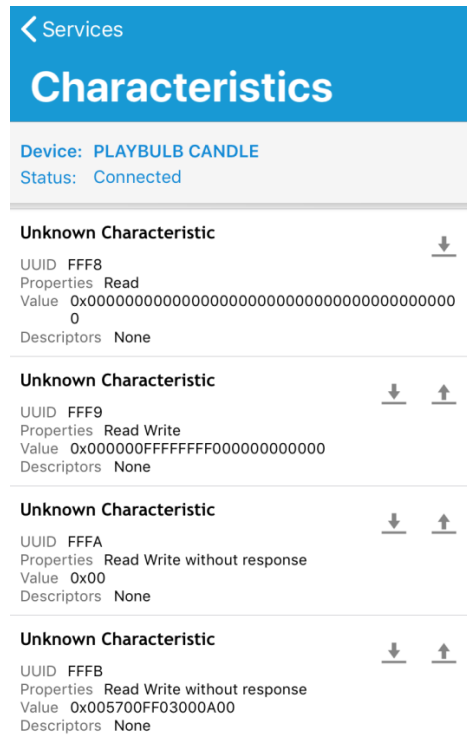


Figura 69. nRFConnect Características

El dispositivo BLE *sniffer* será el correspondiente al modelo *CC2540 USB Dongle* de la empresa *Texas Instruments* [34], mostrado en la Figura 70. Este dispositivo viene pre-programado con un software que permite utilizarlo como un dispositivo *Bluetooth* de captura de paquetes BLE, posibilitando su visualización gráfica mediante el software *Texas Instruments Packet Sniffer* [35].



Figura 70. Dispositivo CC2540 USB Dongle

Una vez que se haya establecido la conexión entre el *smartphone* y el dispositivo *PLAYBULB Candle*, es hora de pasar a apagar el LED y seleccionar los colores rojo, verde y azul para conocer el valor de su trama, estando todas estas opciones disponibles en la aplicación *PLAYBULBX* desde el momento que se ha establecido una conexión, tal y como se muestra en la Figura 71.

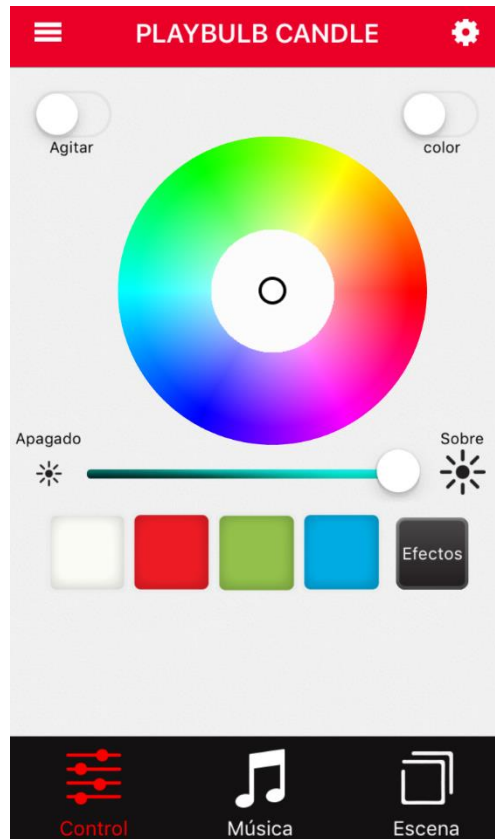


Figura 71. Aplicación PLAYBULBX

Al iniciar la aplicación *Texas Instruments Packet Sniffer*, ésta comienza a detectar todos los paquetes de *advertising* provenientes del dispositivo *PLAYBULB Candle*. Al establecerse la conexión entre el dispositivo *PLAYBULB Candle* y la aplicación móvil, *PLAYBULB Candle* dejará de enviar paquetes de *advertising* y comenzará a establecerse una conexión entre ambos dispositivos, tal y como se muestra en la Figura 72.

P.nbr.	Time (us)	Channel	Access Address	Adv PDU Type	Adv PDU Header	AdvA	AdvData	CRC	RSSI (dBm)	FCS		
7556	+24593 =1127911274	0x25	0x8E99BED6	ADV_IND	Type TxAdd RxAdd PDU-Length 0 0 0 27	0xE09D4B15ACE6	02 01 06 03 03 02 FF 03 19 40 03 02 0A FE 06 FF 4D 49 50 4F 57	0x81C86E	-50	OK		
7557	+446 =1127911720	0x25	0x8E99BED6	ADV_CONNECT_REQ	Type TxAdd RxAdd PDU-Length 5 1 0 34	0x68C9750259BC	0xE09D4B15ACE6 AccessAddr CRCInit WinSize WinOffset Interval 0xAF9A9A6B 7A 1C BB 03 0x0013 0x0018					
7558	+26588 =1127938308	0x09	0xAF9A9A6B	M->S	OK	Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 0 0 1 0	0x20CE7B	-31	OK		
7559	+230 =1127938538	0x09	0xAF9A9A6B	S->M	OK	Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	0x20D3E4	-46	OK		
7560	+230 =1127938768	0x09	0xAF9A9A6B	M->S	Unexp. NESN	Control	Data Header LLID NESN SN MD PDU-Length 3 0 1 0 6	LL_Opcode Version_Ind(0x0C)	LL_Version_Ind VersionNr CompId SubVersNr 0x08 0x000F 0x4109	0xCCECE6	-30	OK

Figura 72. Paquetes BLE estableciendo conexión

Al apagar el LED del *PLAYBULB Candle* desde la aplicación *PLAYBULBX*, la trama que el dispositivo *BLE Central* envía al dispositivo *BLE Peripheral* es la que aparece resaltada en la Figura 73. De la cual se puede obtener que el valor del atributo correspondiente al apagado del LED es 0x00000000.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
							LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
171703	=4590372073	0x05	0xAF9AB35D	M->S	OK	L2CAP-S	2	1	0	1	11	0x0007	0x0004	0x52	0x0019	00 00 00 00	0x313AB1	-30	OK
171704	+318	0x05	0xAF9AB35D	S->M	OK	Empty PDU	1	1	1	0	0	0x1538A	-48	OK					
171705	=4590372621	0x05	0xAF9AB35D	M->S	OK	L2CAP-S	2	0	1	0	11	0x0007	0x0004	0x52	0x0019	00 00 00 00	0x3629D9	-30	OK
171706	+318	0x05	0xAF9AB35D	S->M	OK	Empty PDU	1	0	0	0	0	0x158FF	-48	OK					
171707	+29136	0x0A	0xAF9AB35D	M->S	OK	Empty PDU	1	1	0	0	0	0x15E2C	-30	OK					

Figura 73. Paquetes BLE con trama de apagado del LED del *PLAYBULB Candle*

Para encender el LED se puede utilizar cualquier atributo con valor diferente de 0x00000000, por tanto, el valor que se ha elegido para encender el LED por medio de comandos de voz en la solución final es 0xFFFFFFFF. Al cambiar el color actual del LED a rojo, la trama que el dispositivo *BLE Central* envía al dispositivo *BLE Peripheral* es la que aparece resaltada en la Figura 74, de la cual se puede deducir que el valor del atributo para seleccionar el color rojo del LED es 0x00FF0000.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header					CRC	RSSI (dBm)	FCS					
							LLID	NESN	SN	MD	PDU-Length								
245927	+318	0x15	0x50655D51	S->M	OK	Empty PDU	1	0	0	0	0	0xB311BB	-51	OK					
245928	+29684	0x24	0x50655D51	M->S	OK	L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x52	0x0019	00 FF 00 00	0x006C9D	-30	OK
245929	+318	0x24	0x50655D51	S->M	OK	Empty PDU	1	1	1	0	0	0xB311ACE	-46	OK					
245930	+29684	0x0E	0x50655D51	M->S	OK	Empty PDU	1	0	1	0	0	0xB311C1D	-30	OK					
245931	+229	0x0E	0x50655D51	S->M	OK	Empty PDU	1	0	0	0	0	0xB3111BB	-46	OK					

Figura 74. Paquetes BLE con trama de cambio de color del LED a rojo

Cuando se modifica el color actual del LED a verde, la trama que el dispositivo *BLE Central* envía al dispositivo *BLE Peripheral* es la que aparece resaltada en la Figura 75, de la cual se puede extraer que en este caso el valor del atributo para seleccionar el color verde del LED es 0x0000FF00.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS	
							LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
269161	+30002 =6545308870	0x1F	0x50655D51	M->S	Unexp. NESN	L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x52	0x0019	00 00 FF 00	0x504F12	-30	OK
269162	+318 =6545309188	0x1F	0x50655D51	S->M	OK	Empty PDU	1	1	1	0	0	0xB31ACE					0x504F12	-47	OK
269163	+29684 =6545338872	0x09	0x50655D51	M->S	OK	Empty PDU	1	0	1	0	0	0xB31C1D					0x504F12	-30	OK
269164	+230 =6545339102	0x09	0x50655D51	S->M	OK	Empty PDU	1	0	0	0	0	0xB311BB					0x504F12	-43	OK
269165	+29772 =6545368874	0x18	0x50655D51	M->S	OK	Empty PDU	1	1	0	0	0	0xB31768					0x504F12	-30	OK

Figura 75. Paquetes BLE con trama de cambio de color del LED a verde

Por último, la trama que el dispositivo BLE *Central* manda al dispositivo BLE *Peripheral* cuando se desea cambiar el color del LED actual a azul, es la que aparece resaltada en la Figura 76, de la cual se puede extraer que el valor del atributo para seleccionar el color azul del LED es 0x000000FF.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS						
							LLID	NESN	SN	MD	PDU-Length								
274433	+317 =6624844317	0x15	0x50655D51	S->M	OK	Empty PDU	1	0	0	0	0	0xB311BB		-46	OK				
274434	+29685 =6624874002	0x24	0x50655D51	M->S	OK	L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x52	0x0019	00 00 00 FF	0xAEC07E	-30	OK
274435	+317 =6624874319	0x24	0x50655D51	S->M	OK	Empty PDU	1	1	1	0	0	0xB31ACE					0xAEC07E	-44	OK
274436	+29684 =6624904003	0x0E	0x50655D51	M->S	OK	Empty PDU	1	0	1	0	0	0xB31C1D					0xAEC07E	-31	OK
274437	+230 =6624904233	0x0E	0x50655D51	S->M	OK	Empty PDU	1	0	0	0	0	0xB311BB					0xAEC07E	-43	OK

Figura 76. Paquetes BLE con trama de cambio de color del LED a azul

De la Figura 73, la Figura 74, la Figura 75 y la Figura 76 es importante destacar que en todos los casos se utiliza el mismo identificador de la característica, y por lo tanto, todas utilizan la misma característica. Ahora que se conoce el valor de los atributos para encender, apagar y cambiar el color actual del LED a rojo, verde o azul en el dispositivo *PLAYBULB Candle*, es el momento perfecto para utilizar nuevamente la aplicación *nRFConnect* con el propósito de determinar cuál de las características del servicio primario es la que permite realizar todas estas acciones. Al conocer el tamaño del valor del atributo que permite estos cambios, se pueden descartar aquellas características cuyo tamaño no se ajuste. De este modo, se han reducido las opciones totales de nueve características posibles, a tan solo dos, siendo estas las que poseen los UUID FFFC y FFFD, tal y como se muestra en la Figura 77.

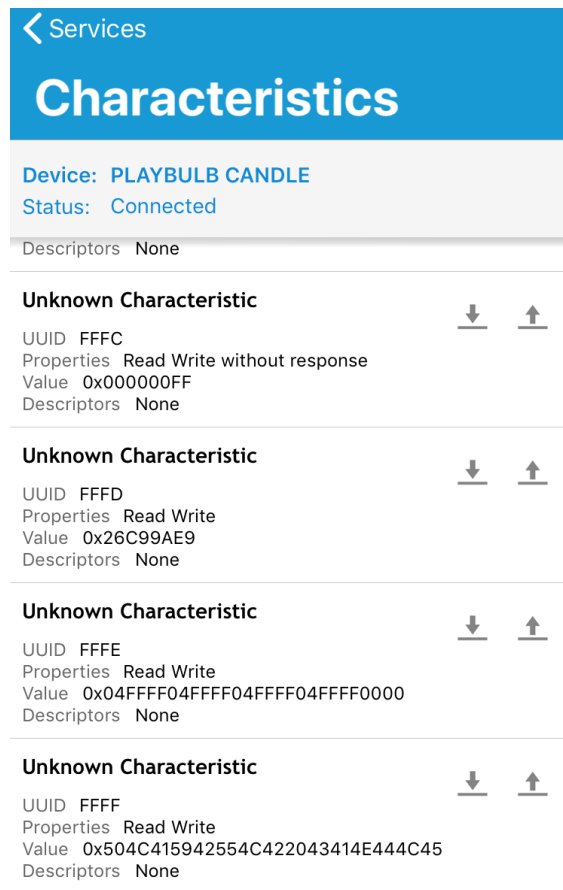


Figura 77. Identificación de características con el nRFConnect

Al escribir el valor 0x00FF0000 en la característica con UUID FFFC, se selecciona el color del LED actual a rojo. Por tanto, se puede concluir que esta característica es la que se encarga de modificar el estado del color del LED. Ahora solo falta por identificar la característica que permite el envío de notificaciones, siendo esta característica fácil de reconocer porque posee las propiedades de lectura (*read*) y notificación (*notify*). De todos los servicios que contiene el dispositivo *PLAYBULB Candle* solo hay una característica que cumpla con estas condiciones, y es la tercera característica del servicio denominado *Unknown Service*, con UUID 0x00001016D10211E19B2300025B00A5A5, tal y como se muestra en la Figura 78. Este servicio se denominará servicio secundario.

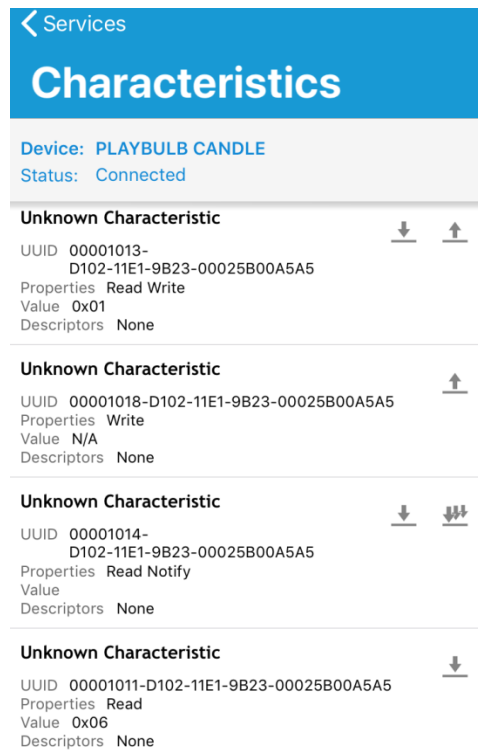


Figura 78. Identificando la característica de notificaciones con nRFConnect

8.2 Alexa Skill Interface

La interacción con el usuario por medio de la voz juega un papel fundamental en la solución final desarrollada en este TFG, por lo que en este apartado se definen las características que tendrá la *skill interface* de Alexa.

El *invocation name* utilizado para invocar la *skill* será *DUO Candle*. El *Intent Schema* de esta solución será el que se recoge en la Figura 79.

```

1  {
2    "intents":
3    [
4      {
5        "intent": "setLightState",
6        "slots": [
7          {
8            "name": "state",
9            "type": "LIST_OF_STATES"
10         }
11       ]
12     },
13     {
14       "intent": "setLightColor",
15       "slots": [
16         {
17           "name": "color",
18           "type": "LIST_OF_SELECTIONS"
19         }
20       ]
21     }
22   ]
23 }

```

Figura 79. Intent Schema de la solución final.

El *intent setLightState* será el encargado de encender (*on*) o apagar (*down*) el LED del dispositivo *PLAYBULB Candle*, utilizando este *intent* el *slot* denominado *state*, que tiene dos estados asociados: *on* y *down*. Por otro lado, el *intent setLightColor* será el encargado de cambiar el color del LED a rojo (*red*), verde (*green*) o azul (*blue*) en el dispositivo *PLAYBULB Candle*. Su *slot* es *color* y está asociado a los estados *red*, *green* y *blue*.

Los *utterances* que se utilizarán para referirse a los *intent* serán los siguientes:

setLightState turn {state}

setLightColor turn {color}

8.3 Alexa Skill Service

Para que la comunicación de los diferentes dispositivos con los servidores de Amazon se realice correctamente, es necesario definir el funcionamiento de la *skill service* de Alexa. En este apartado, solo se incluirán las diferencias presentes en la *skill service* de la solución final con respecto a la solución inicial. Dichas diferencias solo se encuentran en el archivo *index.js*, tal y como se muestra en la Figura 80.

```
--
36 ParticleSkill.prototype = Object.create(AlexaSkill.prototype);
37 ParticleSkill.prototype.constructor = ParticleSkill;
38
39 ParticleSkill.prototype.eventHandlers.onSessionStarted = function (sessionStartedRequest, session) {
40     console.log("ParticleSkill onSessionStarted requestId: " + sessionStartedRequest.requestId
41         + ", sessionId: " + session.sessionId);
42     // any initialization logic goes here
43 };
44
45 ParticleSkill.prototype.eventHandlers.onLaunch = function (launchRequest, session, response) {
46     console.log("ParticleSkill onLaunch requestId: " + launchRequest.requestId + ", sessionId: " + session.sessionId);
47     var speechOutput = "Welcome to Duo Candle. You can ask me to turn on or off the candle. You can also ask me to turn red, green
48         or blue the candle";
49     var repromptText = "You can ask me to turn the light on or off.";
50     response.ask(speechOutput, repromptText);
51 };
52 ParticleSkill.prototype.eventHandlers.onSessionEnded = function (sessionEndedRequest, session) {
53     console.log("ParticleSkill onSessionEnded requestId: " + sessionEndedRequest.requestId
54         + ", sessionId: " + session.sessionId);
55     // any cleanup logic goes here
56 };
```

Figura 80. *Index.js* solución final (I)

La línea 47 de la sección de código representada en la Figura 80 recoge parte de las diferencias presentes en el *index.js*. En este caso, se trata de un cambio en el discurso de respuesta por parte

de Alexa cuando el usuario invoca una *skill*, sin proporcionar un *intent* específico. El resto de las diferencias se incluyen en la Figura 81 y en la Figura 82.

```

58 ParticleSkill.prototype.intentHandlers = {
59   // register custom intent handlers
60   "setLightState": function (intent, session, response) {
61     var slot = intent.slots.state;
62     var slotValue = slot ? slot.value : "";
63
64     if(slotValue) {
65       var fnPr = particle.callFunction({
66         deviceId: PARTICLE_DEVICE_ID,
67         name: slot.name,
68         argument: slotValue,
69         auth: PARTICLE_ACCESS_TOKEN
70       });
71
72       fnPr.then(
73         function(data) {
74           console.log('Function called succesfully:', data);
75
76           var speechOutput = "The candle is now " + slotValue;
77           response.tellWithCard(speechOutput, "duo candle", speechOutput);
78         }, function(err) {
79           console.log('An error occurred:', err);
80         });
81     }
82     else {
83       response.tell("Sorry, I didn't catch what you said.");
84     }
85   },

```

Figura 81. Index.js solución final (II)

Las líneas 60-62 de la Figura 81 representan el cambio en los *intent* y *slots* utilizados. Del mismo modo, las líneas 76 y 77 recogen los cambios realizados en el discurso de respuesta de Alexa, cuando el usuario invoca una *skill* con un *intent* específico.

```

86   "setLightColor": function (intent, session, response) {
87     var slot = intent.slots.color;
88     var slotValue = slot ? slot.value : "";
89
90     if(slotValue) {
91       var fnPr = particle.callFunction({
92         deviceId: PARTICLE_DEVICE_ID,
93         name: slot.name,
94         argument: slotValue,
95         auth: PARTICLE_ACCESS_TOKEN
96       });
97
98       fnPr.then(
99         function(data) {
100           console.log('Function called succesfully:', data);
101
102           var speechOutput = "The candle is now " + slotValue;
103           response.tellWithCard(speechOutput, "duo candle", speechOutput);
104         }, function(err) {
105           console.log('An error occurred:', err);
106         });
107     }
108     else {
109       response.tell("Sorry, I didn't catch what you said.");
110     }
111   },
112 };

```

Figura 82. Index.js solución final (III)

Las líneas 86-88 de la sección de código mostrada en la Figura 82 contienen los cambios en los *intent* y *slots* utilizados. De esta manera, las líneas 102 y 103 recogen los cambios realizados en el discurso de respuesta de Alexa, cuando el usuario invoca una *skill* con un *intent* específico.

Es imprescindible destacar que a todos estos cambios hay que añadir la variación en el código de las variables *APP_ID* y *PARTICLE_DEVICE_ID*, ya que son las encargadas de almacenar el valor del identificador de la *skill* de Alexa y del dispositivo conectando a los servidores de Particle, respectivamente.

8.4 Firmware del dispositivo DUO

Una vez que se ha definido la manera en la que se va a realizar la interacción con el usuario por medio de comandos de voz, y cómo se van a comunicar los diferentes dispositivos con los servidores de Amazon, solo falta definir el *firmware* que va a utilizar el dispositivo *DUO* para comunicarse con *PLAYBULB Candle*.

Para el desarrollo del *firmware* se ha partido del modelo de referencia proporcionado en el repositorio *Github* por la empresa *RedBear*. En dicho modelo, denominado *SimpleBLECentral.ino*, se establecen las bases que permiten a un dispositivo IoT, utilizando el rol de *BLE Central*, establecer una comunicación estable con otro dispositivo que utilice el rol de *BLE Peripheral*. A lo largo de este apartado, se va a definir el funcionamiento del *firmware* desarrollado en el presente TFG apoyándose en el uso de imágenes que contengan secciones del código final, tal y como se muestra en la Figura 83.

```
1 #include "application.h"
2
3
4 #define BLE_SCAN_TYPE      0x00!
5 #define BLE_SCAN_INTERVAL  0x0050!
6 #define BLE_SCAN_WINDOW    0x0030!
```

Figura 83. Firmware del dispositivo DUO (I)

La línea 1 representa la llamada a un archivo de cabecera que es necesario para el correcto funcionamiento del código fuente, ya que se está trabajando con un dispositivo que no pertenece a la empresa *Particle*, por tanto, este archivo se encargará de añadir las funcionalidades necesarias para que las funciones relativas a la interacción con *Particle Cloud* sean compatibles.

Una de las principales funciones de inicialización de BLE es la función *setScansParams*, la cual se encarga de indicar cuáles serán los parámetros de la función de escaneo previa a una comunicación BLE. Para ello, esta función necesita que se declaren tres parámetros, que son los

que aparecen en las líneas 4, 5 y 6, y pueden tomar los siguientes valores dependiendo del tipo de escaneo que se desee realizar:

- *BLE_Scan_Type*:
0x00: Este valor indica escaneo pasivo, es decir, no se enviarán paquetes de solicitud de escaneo, siendo el valor predeterminado.
0x01: Escaneo activo, los paquetes de solicitud de escaneo deben ser enviados para poder realizar una exploración.
0x02 - 0xFF: Reservado para uso futuro.

- *BLE_Scan_Interval*: se define como el intervalo de tiempo desde que el controlador inició su última exploración BLE hasta que comienza la siguiente exploración BLE.
Rango: 0x0004 a 0x4000.
Predeterminado: 0x0010 (10 ms).
Tiempo = $N * 0.625$ mseg.
Rango de tiempo: 2.5 ms a 10.24 segundos.

- *BLE_Scan_Window*: Indica la duración de la exploración BLE. El parámetro *BLE_Scan_Window* debe ser menor o igual que el *BLE_Scan_Interval*.
Rango: 0x0004 a 0x4000.
Predeterminado: 0x0010 (10 ms).
Tiempo = $N * 0.625$ mseg.
Rango de tiempo: 2.5 ms a 10240 ms.

De este modo, la línea 4 indica que se va a realizar un escaneo pasivo, la línea 5 establece que el tiempo entre escaneos será de 60 milisegundos, mientras que la línea 6 indica que cada escaneo durará 30 milisegundos. Por otro lado, las líneas 14-25 de la Figura 84 representan la definición de una estructura, la cual se utiliza para agrupar un conjunto de datos, posiblemente de diferentes tipos, para hacer más eficiente su gestión.

```
14 ▾ typedef struct {  
15     uint16_t connected_handle;;  
16     uint8_t  addr_type;;  
17     bd_addr_t addr;;  
18 ▾     struct {  
19         gatt_client_service_t service;;  
20 ▾         struct {  
21             gatt_client_characteristic_t chars;;  
22             gatt_client_characteristic_descriptor_t descriptor[2];  
23         } chars[9];  
24     } service[2]; |  
25 } Device_t;
```

Figura 84. Firmware del dispositivo DUO (II)

En este caso, se pretende definir la jerarquía que presentaría un dispositivo BLE que tuviera 2 servicios, pudiendo definir en cada uno hasta 9 características, con la opción de que cada una de ellas pueda tener hasta dos descriptores. Además, esta estructura incluye la variable *connected_handle* para asociarse al valor del identificador cuando se establece una conexión correctamente, y otra variable denominada *addr_type*, utilizada para almacenar el valor de la dirección BLE con la cual poder asociarse al dispositivo.

A pesar de que el dispositivo *PLAYBULB Candle* presenta más de dos servicios en su jerarquía, se ha optado por el uso de una estructura que solo contenga dos servicios, dado que solo se va a hacer hincapié en el servicio primario y el servicio secundario del dispositivo final. En la Figura 85 se declaran las variables globales que se van a utilizar a lo largo del código fuente.

```
32 Device_t device;
33 uint8_t serv_index = 0;
34 uint8_t chars_index = 0;
35 uint8_t desc_index = 0;
36 boolean ble_ok = false;
37 int led1 = 0;
38
39 unsigned long lastmills = 0;
40
41 // Connect handle.
42 static uint16_t connected_id = 0xFFFF;
43
44 // The service uuid to be discovered.
45 static uint8_t service1_uuid[16] = { 0x00,0x00,0xFF,0x02,0x00,0x00,0x10,0x00,0x80,0x00,0x00,0x80,0x5F,0x9B,0x34,0xFB };
46 static uint8_t service2_uuid[16] = { 0x00,0x00,0x10,0x16,0xD1,0x02,0x11,0xE1,0x9B,0x23,0x00,0x02,0x5B,0x00,0xA5,0xA5 };
```

Figura 85. Firmware del dispositivo DUO (III)

La línea 32 representa la declaración de la estructura del dispositivo BLE, mientras que las líneas 33-34 son variables que se van a utilizar para determinar qué servicio, característica o descriptor es el que se está utilizando en ese momento, bien sea para obtener información acerca de él o para escribir en él. La línea 39 concuerda con una variable que se va a utilizar para indicar cuál es el intervalo de tiempo que ha transcurrido desde que se recibió la última notificación. La línea 42 hace referencia al valor que toma el identificador cuando se establece una conexión correctamente, siendo el valor por defecto 0xFFFF, un valor incorrecto para cualquiera de las conexiones. Las líneas 45 y 46 representan los UUIDs de los servicios primarios y secundarios a buscar, respectivamente.

En la Figura 86 se define el funcionamiento de la función *ble_avdata_decode*, la cual se encarga de buscar un dato específico dentro de un dato de *advertising* a partir del tipo del dato a buscar. Para ello, recibe como parámetros *type*, que es el tipo del dato a buscar, *adv_data_length*, que es el tamaño del dato de *advertising* y es un valor fundamental para poder recorrer el dato,

**p_avd_data* que es un puntero que apunta al dato de *advertising* para poder hacer comparaciones y saber si se ha encontrado, o no, el dato que se estaba buscando, **len* que es el puntero en el que se va a guardar el tamaño del dato que se ha buscado, y **p_data_field*, que es el puntero en el que se va a almacenar el dato buscado. Esta función devuelve un 0 si se ha encontrado el dato y un 1 si no se ha encontrado.

```

66 ▽ uint32_t ble_advdata_decode(uint8_t type, uint8_t advdata_len, uint8_t *p_advdata, uint8_t *len, uint8_t *p_field_data) {
67     uint8_t index = 0;
68     uint8_t field_length, field_type;
69
70 ▽ while (index < advdata_len) {
71         field_length = p_advdata[index];
72         field_type = p_advdata[index + 1];
73         Serial.print("    - AVD/SR data decoding -> ad_type: ");
74         Serial.print(field_type, HEX);
75         Serial.print(", length: ");
76         Serial.println(field_length, HEX);
77 ▽ if (field_type == type) {
78             memcpy(p_field_data, &p_advdata[index + 2], (field_length - 1));
79             *len = field_length - 1;
80             return 0;
81         }
82         index += field_length + 1;
83     }.
84     return 1;
85 }

```

Figura 86. Firmware del dispositivo DUO (IV)

La Figura 88 contiene la descripción de una función, denominada *reportCallback*, que es llamada por la función *callback onScanReportCallback* cuando se inicia un escaneo, encargándose de gestionar los nuevos paquetes de *advertising* recibidos, o los paquetes de respuesta de escaneo recibidos por parte del dispositivo BLE *Peripheral*. Esta función contiene un único parámetro denominado **report*, el cual es un puntero asociado a una estructura de datos, tal y como se refleja en la Figura 87.

```

typedef struct{
    uint8_t    peerAddrType;
    bd_addr_t  peerAddr;
    int        rssi;
    uint8_t    advEventType;
    uint8_t    advDataLen;
    uint8_t    advData[31];
} advertisementReport_t;

```

Figura 87. Estructura de datos *advertisementReport_t*

En esta estructura de datos hay presentes variables como *peerAddrType*, que se encarga de indicar el tipo dirección del dispositivo, *peerAddr*, que es la variable que contiene la dirección del dispositivo, *advDataLen*, que es la variable que contiene el valor del tamaño del dato de *advertising*, *advData*, que representa un *buffer* de 31 posiciones en las que se almacenan los datos de *advertising*, *rsi*, que es la variable que se encarga de indicar el valor de la señal recibida en dBm, y *advDataType*, que es una variable que se encarga de indicar el tipo de evento de *advertising*, en función de los distintos valores que tome:

- 0x00: Indica que es no direccionable y conectable.
- 0x01: Este tipo es direccionable y conectable.
- 0x02: Es no direccionable y escaneable.
- 0x03: Indica que no es ni direccionable ni conectable.
- 0x04: Respuesta de escaneo.

```

94 void reportCallback(advertisementReport_t *report) {
95     uint8_t index;
96     Serial.println("");
97     Serial.println(" BLE scan! callback: ");
98     Serial.print("   Advertising event type: ");
99     Serial.println(report->advEventType, HEX);
100    Serial.print("   Peer device address type: ");
101    Serial.println(report->peerAddrType, HEX);
102    Serial.print("   Peer device address: ");
103    for (index = 0; index < 6; index++) {
104        Serial.print(report->peerAddr[index], HEX);
105        Serial.print(" ");
106    }
107    Serial.println(" ");
108
109    Serial.print("   RSSI: ");
110    Serial.println(report->rsst, DEC);
111
112    Serial.print("   Advertising/Scan! response data packet: ");
113    for (index = 0; index < report->advDataLen; index++) {
114        Serial.print(report->advData[index], HEX);
115        Serial.print(" ");
116    }
117    Serial.println(" ");
118
119    uint8_t len;
120    uint8_t adv_name[31];
121
122    // Find complete local name.
123
124    if (0x00 == ble_advdata_decode(0x09, report->advDataLen, report->advData, &len, adv_name)) {
125        Serial.println("");
126        Serial.print("   Complete Local Name: ");
127        Serial.println((const char *)adv_name);
128
129        if (0x00 == memcmp(adv_name, "PLAYBULB CANDLE", min(7, len))) {
130            Serial.println("_____ Found PLAYBULB CANDLE");
131            ble.stopScanning();
132            device.addr_type = report->peerAddrType;
133            memcpy(device.addr, report->peerAddr, 6);
134
135            ble.connect(report->peerAddr, report->peerAddrType);
136        }
137    }
138 }

```

Figura 88. Firmware del dispositivo DUO (V)

Las líneas de código 94-117 de la Figura 88 se encargan de mostrar los datos de la estructura *advertisementReport_t* por el terminal conectado al dispositivo *DUO*. Las líneas 124-128 tienen como objetivo determinar el nombre del dispositivo *Peripheral* que está enviando los paquetes de *advertising*, para lo cual, se apoyan en la función *ble_advdata_decode* definida anteriormente.

En el caso de encontrar el dato asociado al tipo de dato del paquete de *advertising*, se tratará de comparar el valor del dato obtenido con el valor del nombre del dispositivo BLE *Peripheral* al que se pretende conectar. En caso de coincidir nuevamente, se finalizará el proceso de escaneo por medio de la función *ble.stopScanning* (línea 131), y se comenzará a establecer la conexión con el dispositivo BLE *Peripheral* por medio de la función *ble.connect*, tal y como se muestra en la línea

135. La función `ble.connect` tiene como parámetros la dirección del dispositivo al que se va a conectar, y el tipo de la dirección del dispositivo BLE *Peripheral* al que se va a conectar.

Una vez que se ha establecido la conexión con el dispositivo BLE *Peripheral*, comienza a ejecutarse la función `deviceConnectedCallback`, que es la encargada de iniciar el proceso de búsqueda de los servicios primarios, en caso de que la conexión se haya establecido satisfactoriamente, tal y como aparece en la Figura 89. Esta función es llamada por la función `callback onConnectedCallback` cuando el proceso de establecimiento de conexión con otro dispositivo se ha completado. Una función `callback` es una función que recibe como argumento otra función, y la ejecuta.

```
148 void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
149     switch (status) {
150         case BLE_STATUS_OK:
151             Serial.println("");
152             Serial.println("_____ Device connected");
153             connected_id = handle;
154             device.connected_handle = handle;
155             Serial.print("          - Device connected handle: ");
156             Serial.println(connected_id);
157             ble.discoverPrimaryServices(handle);
158             break;
159         default: break;
160     }
161 }
```

Figura 89. Firmware del dispositivo DUO (VI)

Esta función tiene como parámetros `status`, que es una variable que refleja el estado de la conexión, indicando con `BLE_STATUS_OK` que la conexión ha sido satisfactoria, y con `BLE_STATUS_CONNECTION_ERROR`, que indica que ha habido un error a la hora de establecer la conexión. El otro parámetro es `handle`, que es el valor del identificador de la conexión establecida. En el caso que el estado de la conexión establecida sea satisfactorio, se inicia la búsqueda de los servicios primarios por medio de la función `ble.discoverPrimaryServices`, la cual tiene como parámetros el identificador de la conexión establecida. Esta función es la encargada de encadenar la llamada de las funciones que se encargan de buscar los servicios, las características y los descriptores, si los hubiera.

La Figura 90 recoge la siguiente función descrita en el código. En este caso se trata de la función `deviceDisconnectedCallback`, que es la encargada de iniciar nuevamente el proceso de escaneo, una vez que se haya realizado la desconexión del dispositivo *Peripheral*. Tiene como parámetro el identificador de conexión, con el fin de poder reiniciar el valor de las variables que lo contenían en caso de que sigan coincidiendo. Esta función también reinicia los valores de la variable `ble_ok`,

variable cuyo valor es *true* cuando se han descubierto los servicios, características y descriptores del servidor GATT, y de la función *digitalWrite*, que se encarga de encender el LED asociado al pin *D7* del dispositivo *DUO* cuando se han descubierto los servicios, características y descriptores del servidor, a modo de confirmación visual de que se ha alcanzado este objetivo.

```
170 void deviceDisconnectedCallback(uint16_t handle){
171     Serial.println("");
172     Serial.println("_____ Device disconnected");
173     Serial.print("          - Device disconnected handle: ");
174     Serial.println(handle);
175     if (connected_id == handle) {
176         Serial.println("");
177         Serial.println("_____ BLE Central restart scanning!");
178         // Disconnect from remote device, restart to scanning.
179         connected_id = 0xFFFF;
180         ble_ok = false;
181         digitalWrite(Led1, LOW);
182
183         ble.startScanning();
184     }
185 }
```

Figura 90. Firmware del dispositivo *DUO* (VII)

El proceso de escaneo se vuelve a iniciar mediante el uso de la función *ble.startScanning*, que estará escaneando en busca de un dispositivo *Peripheral* al que conectarse, hasta que se llame a la función *ble.stopScanning*. La función *deviceDisconnectedCallback* se ejecutará cuando se active la función *callback onDisconnectedCallback*.

La Figura 91 muestra la función *discoveredServiceCallback*, que es la función que se encarga de descubrir los servicios del dispositivo *Peripheral*. Esta función se ejecuta cuando recibe la llamada de la función *callback onServiceDiscoveredCallback*.

```

196 static void discoveredServiceCallback(BLEStatus_t status, uint16_t con_handle, gatt_client_service_t *service) {
197     uint8_t index;
198     if (status == BLE_STATUS_OK) { // Found a service.
199         Serial.println(" ");
200         Serial.println("* Service found successfully");
201         Serial.print(" - Service start handle: ");
202         Serial.println(service->start_group_handle, HEX);
203         Serial.print(" - Service end handle: ");
204         Serial.println(service->end_group_handle, HEX);
205         Serial.print(" - Service uuid16: ");
206         Serial.println(service->uuid16, HEX);
207         Serial.print(" - Service uuid128: ");
208         for (index = 0; index < 16; index++) {
209             Serial.print(service->uuid128[index], HEX);
210             Serial.print(" ");
211         }
212         Serial.println(" ");
213         if (0x00 == memcmp(service->uuid128, service1_uuid, 16)) {
214             Serial.println(" - User defined Service1 uuid128 found successfully");
215             device.service[0].service = *service;
216         }
217         if (0x00 == memcmp(service->uuid128, service2_uuid, 16)) {
218             Serial.println(" - User defined Service2 uuid128 found successfully");
219             device.service[1].service = *service;
220         }
221     }
222     else if (status == BLE_STATUS_DONE) {
223         Serial.println(" ");
224         Serial.println("* Discover custom service completed");
225         serv_index = 0;
226         // All service have been found, start to discover characteristics.
227         // Result will be reported on discoveredCharsCallback.
228         ble.discoverCharacteristics(device.connected_handle, &device.service[serv_index].service);
229     }
230 }

```

Figura 91. Firmware del dispositivo DUO (VIII)

La función *discoveredServiceCallback* tiene como parámetros la variable *con_handle*, que almacena el valor del identificador de la conexión que se ha establecido, la variable *status*, la cual también será utilizada por las funciones *discoveredCharsCallback* y *discoveredCharsDescriptorsCallback*, que es del tipo *BLEStatus_t*, y que puede tener los siguientes valores en función del estado de la conexión que se haya establecido:

- *BLE_STATUS_OK*: señala que se ha establecido la conexión correctamente pero no se han descubierto todos los servicios del dispositivo BLE *Peripheral*. Cuando este valor es utilizado por la función *discoveredCharsCallback*, indica que no se han descubierto todas las características del servicio en el que se está buscando. Por otro lado, cuando este valor es utilizado por la función *discoveredCharsDescriptorCallback*, indicando que no se han descubierto todos los descriptores del servicio en el que se está buscando.
- *BLE_STATUS_DONE*: indica que se ha establecido la conexión correctamente, incluyendo el descubrimiento de los servicios del dispositivo BLE *Peripheral*. Cuando este valor es utilizado por la función *discoveredCharsCallback* indica que se han descubierto todas las características del servicio en el que se está buscando. Del mismo modo, cuando este

valor es utilizado por la función *discoveredCharsDescriptorCallback*, indica que se han descubierto todos los descriptores del servicio en el que se está buscando.

- *BLE_STATUS_CONNECTION_TIMEOUT*: este valor indica que se ha sobrepasado el tiempo máximo de establecimiento de conexión.
- *BLE_STATUS_CONNECTION_ERROR*: supone que ha habido un error al tratar de establecer la conexión.
- *BLE_STATUS_OTHER_ERROR*: indica que se ha producido un error de otro tipo.

La función incorpora además la variable *service*, que es una estructura de datos en la que se van a almacenar los valores del servicio descubierto. La jerarquía de esta estructura es la que se recoge en la Figura 92.

```
typedef struct {
    uint16_t start_group_handle;
    uint16_t end_group_handle;
    uint16_t uuid16;
    uint8_t  uuid128[16];
} gatt_client_service_t;
```

Figura 92. Estructura de datos *service*

La variable *start_group_handle* es la variable en la que se va a almacenar el valor del identificador de inicio. La variable *end_group_handle* es la variable en la que se va a almacenar el valor del identificador de finalización. La variable *uuid16* es la variable en la que se va a almacenar el valor del identificador de 4 bytes del servicio, mientras que la variable *uuid128* es la variable en la que se va a almacenar el valor del identificador de 16 bytes del servicio.

De acuerdo a la Figura 91, la función *discoveredServiceCallback* buscará todos los servicios presentes en el servidor GATT apoyándose en el valor de la variable *status*, con el fin de determinar si quedan servicios por descubrir, o no. Tratará de identificar los servicios primario y secundario, comparando los identificadores de los servicios descubiertos con el valor de las variables *service1_uuid* y *service2_uuid*, que indican el valor del servicio primario y secundario respectivamente.

Una vez que la variable *status* ha indicado que se han descubierto todos los servicios, se hará uso de la función *callback ble.discoveredCharacteristic* para identificar las características del servicio

especificado. Para especificar el servicio, la función *ble.discoveredCharacteristic* posee dos parámetros: *con_handle*, que se encarga de indicar el valor del identificador de la conexión BLE establecida, y **service*, que indica el servicio que se va a explorar. Dada la forma en la que se ha programado el código fuente, esta función siempre comenzará a buscar las características del primer servicio, tal y como se puede ver en las líneas 222-228 de la Figura 91. La función de *callback* llamará a la función *discoveredCharsCallback*, la cual es la función encargada de realizar el proceso de búsqueda de las características del servicio indicado en la función *callback*. En la Figura 93 se puede apreciar el código fuente de esta función.

```

243 v static void discoveredCharsCallback(BLEStatus_t status, uint16_t con_handle, gatt_client_characteristic_t *characteristic) {
244     uint8_t index;
245 v     if (status == BLE_STATUS_OK) { // Found a characteristic.
246         Serial.println(" ");
247         Serial.print("* Characteristic for Service ");
248         Serial.print(serv_index, HEX);
249         Serial.print(" found successfully (");
250         Serial.print(chars_index, HEX);
251         Serial.println(")");
252         Serial.print(" - Characteristic start handle: ");
253         Serial.println(characteristic->start_handle, HEX);
254         Serial.print(" - Characteristic end handle: ");
255         Serial.println(characteristic->end_handle, HEX);
256         Serial.print(" - Characteristic value handle: ");
257         Serial.println(characteristic->value_handle, HEX);
258         Serial.print(" - Characteristic properties: ");
259         Serial.println(characteristic->properties, HEX);
260         Serial.print(" - Characteristic uuid16: ");
261         Serial.println(characteristic->uuid16, HEX);
262         Serial.print(" - Characteristic uuid128 : ");
263 v         for (index = 0; index < 16; index++) {
264             Serial.print(characteristic->uuid128[index], HEX);
265             Serial.print(" ");
266         }
267         Serial.println(" ");
268 v         if (chars_index < 9) {
269             device.service[serv_index].chars[chars_index].chars = *characteristic;
270             chars_index++;
271         }
272     }
273 v     else if (status == BLE_STATUS_DONE) {
274         Serial.println(" ");
275         Serial.print("* Discover all characteristics completed for Service ");
276         Serial.println(serv_index);
277 v         if (serv_index < 2) {
278             chars_index = 0;
279             serv_index++;
280             ble.discoverCharacteristics(device.connected_handle, &device.service[serv_index].service);
281         }
282 v         else {
283             chars_index = 0;
284             serv_index = 0;
285             ble.discoverCharacteristicDescriptors(device.connected_handle, &device.service[serv_index].chars[chars_index].chars);
286         }
287     }
288 }

```

Figura 93. Firmware del dispositivo DUO (IX)

La función *discoveredCharsCallback* tiene como parámetros *status* y *con_handle*, que tienen el mismo propósito que los correspondientes parámetros *status* y *con_handle* de la función *discoveredServiceCallback*. Además, la función *discoveredCharsCallback* también posee el parámetro **characteristic*, el cual es una estructura de datos con una jerarquía y una

funcionalidad muy similar al de la estructura **service*, estructura definida en la función *discoveredServiceCallback*, cuya diferencia radica en que esta nueva estructura posee dos variables adicionales, denominadas *properties* y *value_handle*, tal y como se puede apreciar en la Figura 94.

```
typedef struct {  
    uint16_t start_handle;;  
    uint16_t value_handle;;  
    uint16_t end_handle;;  
    uint16_t properties;;  
    uint16_t uuid16;;  
    uint8_t uuid128[16];;  
} gatt_client_characteristic_t;;
```

Figura 94. Estructura de datos *characteristic*

La variable *properties* se encarga de almacenar el valor de las propiedades de la característica en cuestión, mientras que la variable *value_handle* almacena el valor del identificador de la característica. Por otro lado, la función *discoveredCharsCallback* tratará de buscar todas las características del servicio primario, apoyándose en el valor de la variable *status* para determinar si quedan características por descubrir dentro de este servicio, o no. Una vez que la variable *status* indique que ya no quedan más características por descubrir dentro del servicio primario, se pasará a realizar la búsqueda de las características del servicio secundario, tal y como se muestra en las líneas 273-281 de la Figura 94, repitiendo el mismo procedimiento.

Desde el momento en el que la variable *status* indique que ya no quedan más características por descubrir dentro del servicio secundario, se procederá a inicializar el proceso de búsqueda de los descriptores, para lo cual se hará uso de la función *callback ble.discoveredCharacteristicDescriptor* para identificar los descriptores de cada característica que se encuentra dentro del servicio especificado, si los hubiera.

La función *ble.discoveredCharacteristicDescriptor* posee dos parámetros: *con_handle*, que se encarga de indicar el valor del identificador de la conexión BLE establecida, y **characteristic*, que indica la característica que se va a explorar. Dada la forma en la que se ha programado el código fuente, esta función siempre comenzará a buscar los descriptores de la primera característica del primer servicio, tal y como se puede ver en las líneas 282-285 de la Figura 93. La función de *callback* llamará a la función *discoveredCharsDescriptorsCallback*, la cual se encargará de llevar el proceso de búsqueda de los descriptores de la característica del servicio indicado en la función *callback*. En la Figura 95 se puede apreciar el código fuente de esta función.

```

299 v static void discoveredCharsDescriptorsCallback(BLEStatus_t status, uint16_t con_handle, gatt_client_characteristic_descriptor_t *descriptor) {
300     uint8_t index;
301 v     if (status == BLE_STATUS_OK) { // Found a descriptor.
302         Serial.println(" ");
303         Serial.print(" Descriptor found successfully ");
304         Serial.print(desc_index, HEX);
305         Serial.print(" - Service ");
306         Serial.print(serv_index, HEX);
307         Serial.print(" - Characteristic ");
308         Serial.print(chars_index, HEX);
309         Serial.println(" :");
310         Serial.print(" - Descriptor handle: ");
311         Serial.println(descriptor->handle, HEX);
312         Serial.print(" - Descriptor uuid16: ");
313         Serial.println(descriptor->uuid16, HEX);
314         Serial.print(" - Descriptor uuid128 : ");
315 v         for (index = 0; index < 16; index++) {
316             Serial.print(descriptor->uuid128[index], HEX);
317             Serial.print(" ");
318         }
319         Serial.println(" ");
320 v         if (desc_index < 2) {
321             device.service[serv_index].chars[chars_index].descriptor[desc_index++] = *descriptor;
322         }
323     }
324 v     else if (status == BLE_STATUS_DONE) {
325         // finish.
326 v         if (desc_index > 0) {
327             Serial.println(" ");
328             Serial.print(" Discover all descriptors completed for Service ");
329             Serial.print(serv_index);
330             Serial.print(" - Characteristic ");
331             Serial.println(chars_index);
332         }
333         chars_index++;
334 v         if (chars_index < 9) {
335             desc_index = 0;
336             ble.discoverCharacteristicDescriptors(device.connected_handle, &device.service[serv_index].chars[chars_index].chars);
337         }
338 v         else {
339             chars_index = 0;
340             serv_index++;
341 v             if (serv_index < 2) {
342                 ble.discoverCharacteristicDescriptors(device.connected_handle, &device.service[serv_index].chars[chars_index].chars);
343             }
344 v             else {
345                 serv_index = 0;
346                 ble_ok = true;
347                 digitalWrite(led1, HIGH);
348             }
349         }
350     }
351 }

```

Figura 95. Firmware del dispositivo DUO (X)

La función *discoveredCharsDescriptorsCallback* tiene como parámetros *status* y *con_handle*, que tienen el mismo propósito que los correspondientes parámetros *status* y *con_handle* de las funciones *discoveredServiceCallback* y *discoveredCharsCallback*, respectivamente. Además, la función *discoveredCharsDescriptorsCallback* también posee el parámetro **descriptor*, el cual es una estructura de datos con una jerarquía y una funcionalidad muy similar al de las estructuras **service* y **characteristic*, estructuras definidas en las funciones *discoveredServiceCallback* y *discoveredCharsCallback*, respectivamente, cuya diferencia radica en que esta nueva estructura no posee las variables denominadas *start_handle* y *end_handle*, pero sí que posee una variable adicional denominada *handle*, tal y como se puede apreciar en la Figura 96.

```

typedef struct {
    uint16_t handle;
    uint16_t uuid16;
    uint8_t uuid128[16];
} gatt_client_characteristic_descriptor_t;

```

Figura 96. Estructura de datos descriptor

La variable *handle* se encarga de almacenar el valor del gestor del descriptor. Por otro lado, la función *discoveredCharsDescriptorCallback* tratará de buscar todos los descriptores de la primera característica del servicio primario, apoyándose en el valor de la variable *status* para determinar si quedan descriptores por descubrir. Una vez que la variable *status* indique que ya no quedan más descriptores por descubrir dentro de la primera característica del servicio primario, se pasará a comprobar que no quedan más características en las que realizar la búsqueda de los descriptores dentro del servicio primario. Al comprobar que hay más características se vuelve a comenzar con el proceso de descubrimiento de descriptores. Este procedimiento se repetirá hasta que se hayan explorado todas las características del servicio primario en busca de descriptores, tal y como se describe en las líneas 299-337 de la Figura 95.

Desde el momento en el que se hayan encontrado todos los descriptores de cada una de las características del servicio primario, se pasará a llevar a cabo el mismo procedimiento para hallar los descriptores de cada una de las características del servicio secundario, tal y como se muestra en las líneas 338-343 de la Figura 95. Una vez finalizada la búsqueda de los descriptores en el servicio secundario, se cambiará el valor de la variable *ble_ok* al valor *true*, y además, también se cambiará el valor del LED asociado al pin *D7* del dispositivo DUO de *LOW* a *HIGH*, para indicar en ambos casos que ya se ha terminado el proceso de exploración por parte del cliente GATT hacia los servicios, características y descriptores del servidor GATT, tal y como se expone en las líneas 344-351 de la Figura 95.

Una vez que se han explicado las funciones que permiten al dispositivo BLE *Central* llevar a cabo el proceso de búsqueda de los servicios, características y descriptores del dispositivo BLE *Peripheral*, el siguiente paso será describir las funciones que se encargan del proceso de lectura y escritura de datos. Existen dos funciones de lectura, una para leer el valor de una característica específica, y otra para leer el valor de un descriptor en concreto. La función *readValue* es la encargada de leer el valor de una característica, para poder llevar a cabo esta funcionalidad tiene como parámetros *con_handle*, que es el identificador de la conexión establecida, y *characteristic*, que es la característica que contiene el valor que se va a leer.

Al ejecutarse esta función de lectura se invoca a la función *callback onGattCharacteristicReadCallback*, la cual se encarga de ejecutar la función *gattReadCallback* que

es la que se ocupa de gestionar el proceso de lectura. La Figura 97 muestra el código fuente de esta función.

```
366 void gattReadCallback(BLEStatus_t status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint16_t length) {
367     uint8_t index;
368     if (status == BLE_STATUS_OK) {
369         Serial.println(" ");
370         Serial.println("* Read characteristic value successfully!");
371         Serial.print(" - Connection handle: ");
372         Serial.println(con_handle, HEX);
373         Serial.print(" - Characteristic value attribute handle: ");
374         Serial.println(value_handle, HEX);
375         Serial.print(" - Characteristic value : ");
376         for (index = 0; index < length; index++) {
377             Serial.print(value[index], HEX);
378             Serial.print(" ");
379         }
380         Serial.println("");
381     }
382     else if (status != BLE_STATUS_DONE) {
383         Serial.println(" ");
384         Serial.println("!! Read characteristic value FAILED!");
385         Serial.println(" ");
386     }
387 }
```

Figura 97. Firmware del dispositivo DUO (XI)

La función *gattReadCallback* tiene como parámetros *value_handle*, que es la variable en la que se almacena el valor del identificador del atributo que contiene el valor actual de la característica, **value*, en la que se almacena los datos que se han leído, y *length*, que es la variable utilizada para almacenar el valor del tamaño del dato que se ha leído. Además, esta función también tiene como parámetros *status* y *con_handle*, los cuales han sido definidos anteriormente para otras funciones.

Si el parámetro *status* indica que la lectura se ha realizado correctamente, la función *gattReadCallback* mostrará por pantalla el identificador de la conexión, el identificador del atributo que contiene el valor actual de la característica, y el valor de la característica que acaba de leer. En el caso de que el parámetro *status* indique que la lectura ha fallado, la función mostrará por pantalla un mensaje indicando que ha habido un error en la lectura de la característica.

La función encargada de leer el valor de un descriptor se denomina *readDescriptorValue*. Para que esta función pueda llevar a cabo esta funcionalidad, tiene como parámetros *con_handle*, que es el identificador de la conexión establecida, y *descriptor*, que es el descriptor que contiene el valor que se va a leer. Al ejecutarse esta función de lectura se invoca a la función *callback onGattDescriptorReadCallback*, la cual se encarga de ejecutar la función

gattReadDescriptorCallback, que es la que se ocupa de gestionar el proceso de lectura del descriptor. La Figura 98 muestra el código fuente de esta función.

```
422 void gattReadDescriptorCallback(BLEStatus_t status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint16_t
length) {
423     uint8_t index;
424     if(status == BLE_STATUS_OK) {
425         Serial.println("");
426         Serial.print("D --- gattReadDescriptorCallback (");
427         Serial.println(" ");
428         Serial.println("* Read descriptor value successfully:");
429         Serial.print("  - Connection handle: ");
430         Serial.println(con_handle, HEX);
431         Serial.print("  - Descriptor value attribute handle: ");
432         Serial.println(value_handle, HEX);
433         Serial.print("  - Descriptor value :: ");
434         for (index = 0; index < length; index++) {
435             Serial.print(value[index], HEX);
436             Serial.print(" ");
437         }
438         Serial.println(" ");
439         if (value_handle == device.service[1].chars[0].descriptor[0].handle) {
440             if ((value[0] == 0x01) && (value[1] == 0x00)) {
441                 Serial.println(" ");
442                 Serial.println("____ Notifications enabled");
443             }
444             else if ((value[0] == 0x00) && (value[1] == 0x00)) {
445                 Serial.println(" ");
446                 Serial.println("____ Notifications disabled");
447             }
448         }
449     }
450 }
```

Figura 98. Firmware del dispositivo DUO (XII)

La función *gattReadDescriptorCallback* tiene los mismos parámetros que la función *gattReadCallback*, con la diferencia de que lee descriptores en lugar de características. Si el parámetro *status* indica que la lectura se ha realizado correctamente, la función *gattReadCallback* mostrará por pantalla el identificador de la conexión, el identificador del atributo que contiene el valor actual del descriptor, y el valor del descriptor que acaba de leer. Además, esta función también se encarga de comprobar si las funciones están habilitadas o no. En el caso de que el parámetro *status* indique que la lectura ha fallado, la función mostrará por pantalla un mensaje indicando que ha habido un error en la lectura de la característica.

Por otro lado, existen diferentes funciones de escritura de datos, pero en esta solución solo se va utilizar la función denominada *writeValueWithoutResponse*, ya que la característica en la que se pretende escribir el dato que encienda/apague o cambie el color el LED del dispositivo *PLAYBULB Candle*, tiene la propiedad *WRITE_WITHOUT_RESPONSE*. La función *writeValueWithoutResponse* permite escribir un valor en una característica específica, para lo cual tiene como parámetros *con_handle*, que es el identificador de la conexión establecida, *char_value_handle*, que es el

identificador del atributo que contiene el valor actual de la característica, *size*, que es la variable que contiene el valor del tamaño del dato que se va a escribir, y *data*, que es el dato que se va a escribir en la característica.

Al ejecutarse la función *writeValueWithoutResponse* se llama a la función *callback onGattCharacteristicWrittenCallback*, la cual se encarga a su vez de ejecutar la función *gattWrittenCallback* para que gestione la escritura. La Figura 99 muestra el código fuente de esta función.

```
397 void gattWrittenCallback(BLEStatus_t status, uint16_t con_handle) {
398     if (status == BLE_STATUS_DONE) {
399         Serial.println(" ");
400         Serial.println("* Write characteristic value done:");
401         Serial.print("   - Connection handle: ");
402         Serial.println(con_handle, HEX);
403     }
404     else {
405         Serial.println(" ");
406         Serial.println("! Write characteristic value FAILED");
407         Serial.println(" ");
408     }
409 }
```

Figura 99. Firmware del dispositivo DUO (XIII)

Esta función tiene como parámetros las variables *status* y *con_handle*. Si el valor de la variable *status* indica que el proceso de escritura ha sido correcto, la función muestra por pantalla un mensaje indicándolo, mientras que si el proceso de escritura no ha sido correcto, la función muestra un mensaje por pantalla indicándolo.

La siguiente función a describir es la denominada *gattWriteCCDCallback*. Esta función se encarga de gestionar la escritura del valor del CCCD, tal y como se muestra en la Figura 100. La función *gattWriteCCDCallback* se ejecuta cuando recibe la llamada de la función *callback onGattWriteClientCharacteristicConfigCallback*.

```

460 void gattWriteCCDCallback(BLEStatus_t status, uint16_t con_handle) {
461     Serial.println("");
462     Serial.print("D --- gattWriteCCDCallback (");
463     Serial.print(status, HEX);
464     Serial.println(")");
465     if (status == BLE_STATUS_DONE) {
466         Serial.println(" ");
467         Serial.println("* Write CCCD value successfully");
468         Serial.print("  - Connection handle: ");
469         Serial.println(con_handle, HEX);
470         Serial.print("  - CCCD value: ");
471         Serial.println(GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION, HEX);
472         ble.readDescriptorValue(device.connected_handle, device.service[1].chars[0].descriptor[0].handle);
473     }
474     else {
475         Serial.println(" ");
476         Serial.println("! Write CCCD value FAILED");
477     }
478 }

```

Figura 100. Firmware del dispositivo DUO (XIV)

Esta función tiene como parámetros las variables *status* y *con_handle*. La función *gattWriteCCDCallback* se encargará de mostrar por pantalla el resultado del proceso de escritura de la configuración del descriptor de la característica del cliente, siempre y cuando el parámetro *status* haya indicado que la escritura del valor ha sido correcta. Además, también se encargará de ejecutar la lectura del valor del descriptor por medio de la función *readDescriptorValue*. En caso de que la escritura del valor haya fallado, la función mostrará por pantalla un mensaje indicándolo.

La función *gattNotifyUpdateCallback* es la función que se encarga de la gestión de los paquetes de notificación recibidos por parte del dispositivo BLE *Peripheral*. Esta función muestra por pantalla la información referente a la notificación recibida, tal y como se muestra en Figura 101. La función *gattNotifyUpdateCallback* se ejecuta cuando recibe la llamada de la función *callback onGattNotifyUpdateCallback*.

```

492 void gattNotifyUpdateCallback(BLEStatus_t status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint16_t
length) {
493     uint8_t index;
494     Serial.println(" ");
495     Serial.println("* Received new notification (");
496     Serial.print(millis()-lastmills);
497     Serial.println(" ms:");
498     lastmills = millis();
499     Serial.print("  - Connection handle: ");
500     Serial.println(con_handle, HEX);
501     Serial.print("  - Characteristic value attribute handle: ");
502     Serial.println(value_handle, HEX);
503     Serial.print("  - Notified value: ");
504     for (index = 0; index < length; index++) {
505         Serial.print(value[index], HEX);
506         Serial.print(" ");
507     }
508     Serial.println(" ");
509 }

```

Figura 101. Firmware del dispositivo DUO (XV)

Ahora que ya se han descrito todas las funciones asociadas a la funcionalidad del dispositivo *DUO* como BLE *Central*, falta por definir aquellas funciones que permiten al dispositivo *DUO* comunicarse con Alexa para poder ejecutar los comandos indicados por el usuario en su interacción mediante la voz.

La función *setLightState* es la encargada de encender o apagar el LED del dispositivo *PLAYBULB Candle* en función del valor del parámetro que haya recibido. Esta función tiene como parámetro *state*, el cual puede tener los valores *on*, valor con el cual se enciende el LED, y *down*, valor con el cual se apaga el LED. La línea 517 representa la variable que contiene el valor de la trama de 16 bits que enciende el LED, mientras que la línea 518 representa la variable que contiene el valor de la trama de 16 bits que apaga el LED, tal y como se puede ver en la Figura 102.

```
516 int setLightState(String state) {
517     uint8_t data_playbulb_ON[] = {0x00,0xFF, 0xFF, 0xFF};
518     uint8_t data_playbulb_OFF[] = {0x00,0x00, 0x00, 0x00};
519     uint8_t index;
520
521     Serial.println("state: " + state);
522
523     if(state == "on") {
524         Serial.println("");
525         Serial.println("> Service[0]-Characteristic[5]_playbulb_write: ");
526         Serial.println(" - data_playbulb_ON");
527         Serial.print(" - Connection handle: ");
528         Serial.println(device.connected_handle);
529         Serial.print(" - Characteristic value attribute handle: ");
530         Serial.println(device.service[0].chars[5].chars.value_handle);
531         Serial.print(" - Characteristic value : ");
532         for (index = 0; index < sizeof(data_playbulb_ON); index++) {
533             Serial.print(data_playbulb_ON[index], HEX);
534             Serial.print(" ");
535         }
536         ble.writeValueWithoutResponse(device.connected_handle, device.service[0].chars[5].chars.value_handle, sizeof(data_playbulb_ON),
537             data_playbulb_ON);
538         ble.readValue(device.connected_handle,&device.service[0].chars[5].chars);
539     }
540     else if(state == "down") {
541         Serial.println("");
542         Serial.println("> Service[0]-Characteristic[5]_playbulb_write: ");
543         Serial.println(" - data_playbulb_OFF");
544         Serial.print(" - Connection handle: ");
545         Serial.println(device.connected_handle);
546         Serial.print(" - Characteristic value attribute handle: ");
547         Serial.println(device.service[0].chars[5].chars.value_handle);
548         Serial.print(" - Characteristic value : ");
549         for (index = 0; index < sizeof(data_playbulb_OFF); index++) {
550             Serial.print(data_playbulb_OFF[index], HEX);
551             Serial.print(" ");
552         }
553         ble.writeValueWithoutResponse(device.connected_handle, device.service[0].chars[5].chars.value_handle, sizeof(data_playbulb_OFF),
554             data_playbulb_OFF); ble.readValue(device.connected_handle,&device.service[0].chars[5].chars);
555     }
556 }
```

Figura 102. Firmware del dispositivo *DUO* (XVI)

Una vez que se haya determinado si el LED debe encenderse o apagarse, el dispositivo *DUO* transmitirá el dato correspondiente al dispositivo BLE *Peripheral* por medio de la función de

escritura de característica `writeValueWithoutResponse`. Cuando el proceso de escritura finalice, se procederá a leer el valor escrito para verificar que se ha escrito correctamente el dato deseado.

La función `setLightColor` es la encargada de cambiar el color del dispositivo `PLAYBULB Candle`. Esta función tiene como parámetro `color`, el cual puede tener los valores `red` (rojo), `green` (verde) o `blue` (azul), en función del color que se vaya a seleccionar en el LED. La Figura 103 y la Figura 104 recogen el código fuente de esta función

```
556 ▼ int setLightColor(String color) {
557     uint8_t data_playbulb_RED[] = {0x00,0xFF, 0x00, 0x00};
558     uint8_t data_playbulb_GREEN[] = {0x00,0x00, 0xFF, 0x00};
559     uint8_t data_playbulb_BLUE[] = {0x00,0x00, 0x00, 0xFF};
560     uint8_t index;
561
562     Serial.println("color: " + color);
563
564 ▼ if(color == "red") {
565     Serial.println("");
566     Serial.println("> Service[0]-Characteristic[5]_playbulb_write: ");
567     Serial.println(" - data_playbulb_RED");
568     Serial.print(" - Connection handle: ");
569     Serial.println(device.connected_handle);
570     Serial.print(" - Characteristic value attribute handle: ");
571     Serial.println(device.service[0].chars[5].chars.value_handle);
572     Serial.print(" - Characteristic value : ");
573 ▼ for (index = 0; index < sizeof(data_playbulb_RED); index++) {
574     Serial.print(data_playbulb_RED[index], HEX);
575     Serial.print(" ");
576     }
577     ble.writeValueWithoutResponse(device.connected_handle, device.service[0].chars[5].chars.value_handle,
578     sizeof(data_playbulb_RED), data_playbulb_RED);
578     ble.readValue(device.connected_handle,&device.service[0].chars[5].chars);
579 }
}
```

Figura 103. Firmware del dispositivo DUO (XVII)

```
580 ▼ else if(color == "green") {
581     Serial.println("");
582     Serial.println("> Service[0]-Characteristic[5]_playbulb_write: ");
583     Serial.println(" - data_playbulb_GREEN");
584     Serial.print(" - Connection handle: ");
585     Serial.println(device.connected_handle);
586     Serial.print(" - Characteristic value attribute handle: ");
587     Serial.println(device.service[0].chars[5].chars.value_handle);
588     Serial.print(" - Characteristic value : ");
589 ▼ for (index = 0; index < sizeof(data_playbulb_GREEN); index++) {
590     Serial.print(data_playbulb_GREEN[index], HEX);
591     Serial.print(" ");
592     }
593     ble.writeValueWithoutResponse(device.connected_handle, device.service[0].chars[5].chars.value_handle,
594     sizeof(data_playbulb_GREEN), data_playbulb_GREEN);
594     ble.readValue(device.connected_handle,&device.service[0].chars[5].chars);
595 }
596 ▼ else if(color == "blue") {
597     Serial.println("");
598     Serial.println("> Service[0]-Characteristic[5]_playbulb_write: ");
599     Serial.println(" - data_playbulb_BLUE");
600     Serial.print(" - Connection handle: ");
601     Serial.println(device.connected_handle);
602     Serial.print(" - Characteristic value attribute handle: ");
603     Serial.println(device.service[0].chars[5].chars.value_handle);
604     Serial.print(" - Characteristic value : ");
605 ▼ for (index = 0; index < sizeof(data_playbulb_BLUE); index++) {
606     Serial.print(data_playbulb_BLUE[index], HEX);
607     Serial.print(" ");
608     }
609     ble.writeValueWithoutResponse(device.connected_handle, device.service[0].chars[5].chars.value_handle,
610     sizeof(data_playbulb_BLUE), data_playbulb_BLUE);
610     ble.readValue(device.connected_handle,&device.service[0].chars[5].chars);
611 }
612 }
```

Figura 104. Firmware del dispositivo DUO (XVIII)

La función `setLightColor` transmitirá el dato correspondiente al dispositivo BLE *Peripheral* por medio de la función de escritura de característica `writeValueWithoutResponse`. Cuando el proceso de escritura finalice, se procederá a leer el valor escrito para verificar que se ha escrito correctamente el dato deseado.

La función `setup` es la primera función en ejecutarse dentro de un programa en el dispositivo DUO. Es, básicamente, donde se inicializan las funciones que llevará a cabo el dispositivo. La Figura 105 muestra el código fuente de esta función.

```
618 void setup() {
619
620     Serial.begin(115200);
621     pinMode(led1, OUTPUT);
622     digitalWrite(led1, LOW);
623     Particle.function("state", setLightState);
624     Particle.function("color", setLightColor);
625
626     delay(5000);
627
628     Serial.println(" ");
629     Serial.println("____ BLE Central <-> PlayBulb ____");
630
631     // Initialize ble_stack.
632     ble.init();
633
634     // Register callback functions.
635     ble.onConnectedCallback(deviceConnectedCallback);
636     ble.onDisconnectedCallback(deviceDisconnectedCallback);
637     ble.onScanReportCallback(reportCallback);
638     ble.onServiceDiscoveredCallback(discoveredServiceCallback);
639     ble.onCharacteristicDiscoveredCallback(discoveredCharsCallback);
640     ble.onDescriptorDiscoveredCallback(discoveredCharsDescriptorsCallback);
641     ble.onGattCharacteristicReadCallback(gattReadCallback);
642     ble.onGattCharacteristicWrittenCallback(gattWrittenCallback);
643     ble.onGattDescriptorReadCallback(gattReadDescriptorCallback);
644     ble.onGattWriteClientCharacteristicConfigCallback(gattWriteCCDCallback);
645     ble.onGattNotifyUpdateCallback(gattNotifyUpdateCallback);
646
647     // Set scan parameters.
648     ble.setScanParams(BLE_SCAN_TYPE, BLE_SCAN_INTERVAL, BLE_SCAN_WINDOW);
649
650     // Start scanning.
651     ble.startScanning();
652     Serial.println("____ BLE Central start scanning");
653 }
```

Figura 105. Firmware del dispositivo DUO (XIX)

Ya que el dispositivo *DUO* va a realizar una comunicación por puerto serie, en la línea 620 se especifica la velocidad en baudios para la transmisión de datos en serie. Se configura el pin *D7* del *DUO* como salida, y se establece a nivel bajo, tal y como se refleja en las líneas 621 y 622. Las

líneas 623 y 624 vinculan las funciones *setLightState* y *setLightColor* a *Particle Cloud* mediante las *particle functions*. La línea 632 habilita la interfaz HCI para la comunicación entre el *host* y el controlador, activando la pila de protocolos de BLE. Las líneas 635-645 se encargan de declarar todas las funciones *callback* utilizadas a lo largo del código fuente. La línea 647 contiene la función *setScanParams*, la cual establece los parámetros de escaneo mediante las variables *BLE_SCAN_TYPE*, *BLE_SCAN_WINDOW* y *BLE_SCAN_INTERVAL* definidas anteriormente. La línea 651 se encarga de iniciar el proceso de escaneo.

La Figura 106 recoge la función *loop*, que se encarga de ejecutar un número infinito de veces el código que se sitúe dentro de su rango. En este caso esta función está en blanco, pero su inclusión en el código es necesaria para la correcta compilación del mismo.

```
657 ▾ /**
658     * @brief: Loop.
659     */
660 ▾ void loop() {
661
662 }
```

Figura 106. Firmware del dispositivo DUO (XX)

8.5 Validación

Con el desarrollo de este apartado se pretende comprobar que el funcionamiento de la solución final desarrollada en el presente TFG es el deseado. Para ello, se ha recopilado a través de imágenes el proceso de respuesta de las funciones que implementa el dispositivo *DUO* mediante la línea de comandos.

```
* BLE scan callback:
  - Advertising event type: 4
  - Peer device address type: 0
  - Peer device address: E0 9D 4B 15 AC E6
  - RSSI: -43
  - Advertising/Scan response data packet: 10 9 50 4C 41 59 42 55 4C 42 20 43 4
1 4E 44 4C 45
  - ADV/SR data decoding -> ad_type: 9, length: 10
  - Complete Local Name: PLAYBULB CANDLE
----- Found PLAYBULB CANDLE
----- Device connected
  - Device connected handle: 64
```

Figura 107. Establecimiento de conexión entre dispositivos

La Figura 107 muestra como el dispositivo BLE *Central* implantado en el *DUO* ha detectado un paquete de *Advertising* por parte de un dispositivo BLE *Peripheral* y cómo se ha conectado con él,

cuando ha detectado por medio del parámetro *Complete Local Name* que es el dispositivo *PLAYBULB Candle* que estaba buscando. La Figura 108 muestra el proceso de identificación de los servicios del dispositivo BLE *Peripheral*.

```
* Service found successfully
- Service start handle: 8
- Service end handle: B
- Service uuid16: 1801
- Service uuid128 : 0 0 18 1 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Service found successfully
- Service start handle: C
- Service end handle: 1F
- Service uuid16: FF02
- Service uuid128 : 0 0 FF 2 0 0 10 0 80 0 0 80 5F 9B 34 FB
- User defined Service1 uuid128 found successfully

* Service found successfully
- Service start handle: 20
- Service end handle: 23
- Service uuid16: 180F
- Service uuid128 : 0 0 18 F 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Service found successfully
- Service start handle: 24
- Service end handle: 30
- Service uuid16: 180A
- Service uuid128 : 0 0 18 A 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Service found successfully
- Service start handle: 31
- Service end handle: FFFF
- Service uuid16: 0
- Service uuid128 : 0 0 10 16 D1 2 11 E1 9B 23 0 2 5B 0 A5 A5
- User defined Service2 uuid128 found successfully

* Discover custom service completed
```

Figura 108. Identificación de los servicios del dispositivo *Peripheral*

Una vez que se han descubierto todos los servicios, comienza el proceso de identificación de las características definidas en los servicios primario y secundario.

```

* Characteristic for Service 0 found successfully (0) :
- Characteristic start handle: D
- Characteristic end handle: F
- Characteristic value handle: E
- Characteristic properties: 10
- Characteristic uuid16: 2A37
- Characteristic uuid128 : 0 0 2A 37 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (1) :
- Characteristic start handle: 10
- Characteristic end handle: 11
- Characteristic value handle: 11
- Characteristic properties: 2
- Characteristic uuid16: FFF8
- Characteristic uuid128 : 0 0 FF F8 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (2) :
- Characteristic start handle: 12
- Characteristic end handle: 13
- Characteristic value handle: 13
- Characteristic properties: A
- Characteristic uuid16: FFF9
- Characteristic uuid128 : 0 0 FF F9 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (3) :
- Characteristic start handle: 14
- Characteristic end handle: 15
- Characteristic value handle: 15
- Characteristic properties: 6
- Characteristic uuid16: FFFA
- Characteristic uuid128 : 0 0 FF FA 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (4) :
- Characteristic start handle: 16
- Characteristic end handle: 17
- Characteristic value handle: 17
- Characteristic properties: 6
- Characteristic uuid16: FFFB
- Characteristic uuid128 : 0 0 FF FB 0 0 10 0 80 0 0 80 5F 9B 34 FB

```

Figura 109. Identificación de las características del servicio primario (I)

La Figura 109 y la Figura 110 reflejan la respuesta de las funciones encargadas de llevar a cabo el proceso de identificación de las características definidas en el servicio primario.

```

* Characteristic for Service 0 found successfully (5) :
- Characteristic start handle: 18
- Characteristic end handle: 19
- Characteristic value handle: 19
- Characteristic properties: 6
- Characteristic uuid16: FFFC
- Characteristic uuid128 : 0 0 FF FC 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (6) :
- Characteristic start handle: 1A
- Characteristic end handle: 1B
- Characteristic value handle: 1B
- Characteristic properties: A
- Characteristic uuid16: FFFD
- Characteristic uuid128 : 0 0 FF FD 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (7) :
- Characteristic start handle: 1C
- Characteristic end handle: 1D
- Characteristic value handle: 1D
- Characteristic properties: A
- Characteristic uuid16: FFFE
- Characteristic uuid128 : 0 0 FF FE 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Characteristic for Service 0 found successfully (8) :
- Characteristic start handle: 1E
- Characteristic end handle: 1F
- Characteristic value handle: 1F
- Characteristic properties: A
- Characteristic uuid16: FFFF
- Characteristic uuid128 : 0 0 FF FF 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Discover all characteristics completed for Service 0

```

Figura 110. Identificación de las características del servicio primario (II)

Por otro lado, la Figura 111 muestra la respuesta de las funciones encargadas de descubrir las características del servicio secundario.

```

* Characteristic for Service 1 found successfully (0) :
- Characteristic start handle: 32
- Characteristic end handle: 33
- Characteristic value handle: 33
- Characteristic properties: A
- Characteristic uuid16: 0
- Characteristic uuid128 : 0 0 10 13 D1 2 11 E1 9B 23 0 2 5B 0 A5 A5

* Characteristic for Service 1 found successfully (1) :
- Characteristic start handle: 34
- Characteristic end handle: 35
- Characteristic value handle: 35
- Characteristic properties: 8
- Characteristic uuid16: 0
- Characteristic uuid128 : 0 0 10 18 D1 2 11 E1 9B 23 0 2 5B 0 A5 A5

* Characteristic for Service 1 found successfully (2) :
- Characteristic start handle: 36
- Characteristic end handle: 38
- Characteristic value handle: 37
- Characteristic properties: 12
- Characteristic uuid16: 0
- Characteristic uuid128 : 0 0 10 14 D1 2 11 E1 9B 23 0 2 5B 0 A5 A5

* Characteristic for Service 1 found successfully (3) :
- Characteristic start handle: 39
- Characteristic end handle: FFFF
- Characteristic value handle: 3A
- Characteristic properties: 2
- Characteristic uuid16: 0
- Characteristic uuid128 : 0 0 10 11 D1 2 11 E1 9B 23 0 2 5B 0 A5 A5

* Discover all characteristics completed for Service 1

* Discover all characteristics completed for Service 2

```

Figura 111. Identificación de las características del servicio secundario

El siguiente paso consiste en descubrir los descriptores de las características definidos en los servicios primario y secundario. La Figura 112 expone la respuesta de las funciones que se encargan de este proceso de identificación.

```

* Descriptor found successfully 0 - Service 0 - Characteristic 0 :
- Descriptor handle: F
- Descriptor uuid16: 2902
- Descriptor uuid128 : 0 0 29 2 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Discover all descriptors completed for Service 0 - Characteristic 0

* Descriptor found successfully 0 - Service 1 - Characteristic 2 :
- Descriptor handle: 38
- Descriptor uuid16: 2902
- Descriptor uuid128 : 0 0 29 2 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Discover all descriptors completed for Service 1 - Characteristic 2
state: down

```

Figura 112. Identificación de los descriptores de los servicios del dispositivo Peripheral

En este momento, ya se han identificado todos los servicios, las características y los descriptores del dispositivo BLE *Peripheral* correspondiente al dispositivo *PLAYBULB Candle*. Por tanto, ya solo queda comprobar el proceso de escritura del dato y la comprobación de que se ha escrito correctamente. Para ello, se ha hecho una recopilación de imágenes que contienen el resultado de lo que las funciones encargadas de escribir muestran por pantalla, mientras se están

ejecutando, tal y como se ve en la Figura 113, donde se muestra el resultado de apagar el LED del dispositivo *PLAYBULB Candle*.

```
state: down
> Service[0]-Characteristic[5]_playbulb_write:
- data_playbulb_OFF
- Connection handle: 64
- Characteristic value attribute handle: 25
- Characteristic value : 0 0 0 0
* Read characteristic value successfully:
- Connection handle: 40
- Characteristic value attribute handle: 19
- Characteristic value : 0 0 0 0
```

Figura 113. Apagado del LED

En la Figura 113 se observa cómo se escribe el valor 0000 para apagar el LED cuando el estado del *slot state* es *down*, y cómo al leer el dato escrito se obtiene el mismo valor que se ha escrito confirmando que el proceso de apagado se ha llevado a cabo satisfactoriamente

La Figura 114 muestra el resultado de encender el LED. En ella se puede apreciar claramente cómo se escribe el valor 0 FF FF FF FF para encender el LED cuando el estado del *slot state* es *on*, y cómo al leer el dato escrito se obtiene el mismo valor que se ha escrito, confirmando igualmente que el proceso de encender el LED se ha llevado a cabo satisfactoriamente

```
state: on
> Service[0]-Characteristic[5]_playbulb_write:
- data_playbulb_ON
- Connection handle: 64
- Characteristic value attribute handle: 25
- Characteristic value : 0 FF FF FF
* Read characteristic value successfully:
- Connection handle: 40
- Characteristic value attribute handle: 19
- Characteristic value : 0 FF FF FF
```

Figura 114. Encendido del LED

La Figura 115 describe el proceso de que llevan a cabo las funciones del dispositivo *DUO* cuando proceden a cambiar el color actual del LED a rojo, verde y azul.

```

color: red

> Service[0]-Characteristic[5]_playbulb_write:
- data_playbulb_RED
- Connection handle: 64
- Characteristic value attribute handle: 25
- Characteristic value ; 0 FF 0 0
* Read characteristic value successfully:
- Connection handle: 40
- Characteristic value attribute handle: 19
- Characteristic value ; 0 FF 0 0
color: green

> Service[0]-Characteristic[5]_playbulb_write:
- data_playbulb_GREEN
- Connection handle: 64
- Characteristic value attribute handle: 25
- Characteristic value ; 0 0 FF 0
* Read characteristic value successfully:
- Connection handle: 40
- Characteristic value attribute handle: 19
- Characteristic value ; 0 0 FF 0
color: blue

> Service[0]-Characteristic[5]_playbulb_write:
- data_playbulb_BLUE
- Connection handle: 64
- Characteristic value attribute handle: 25
- Characteristic value ; 0 0 0 FF
* Read characteristic value successfully:
- Connection handle: 40
- Characteristic value attribute handle: 19
- Characteristic value ; 0 0 0 FF

```

Figura 115. Cambio de color del LED

En la Figura 115 se puede apreciar cómo, según el estado del *slot color* que hayan recibido, *red*, *green* o *blue* en cada caso, escribirán un valor u otro. En cualquiera de los casos se puede apreciar cómo se escribe el valor correcto, y cómo se ha leído el valor esperado. A continuación, se van a incluir imágenes de las tramas capturadas con el dispositivo BLE *sniffer* durante el proceso de escritura, con el objetivo de verificar que los valores que se pretenden escribir se han transmitido correctamente.

La Figura 116 muestra la captura de la trama de envío de escritura que indica que la luz del LED se apagará, siendo esta la trama resaltada, mientras que una trama más abajo se muestra la captura de la trama de solicitud de lectura, para confirmar que se ha escrito el valor correcto en la característica adecuada.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS				
5124	+229	0x16	0xAF9AADD0	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x747B15	-42	OK			
5125	+29772	0x21	0xAF9AADD0	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Write_Command		CRC	RSSI (dBm)	FCS
5126	+317	0x21	0xAF9AADD0	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x747060	-40	OK			
5127	+29682	0x07	0xAF9AADD0	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Read_Req		CRC	RSSI (dBm)	FCS
5128	+286	0x07	0xAF9AADD0	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x747B15	-37	OK			
5129	+28715																

Figura 116. Tramas de apagado del LED

La Figura 117, la Figura 118, la Figura 119 y la Figura 120 muestran la capturas de las tramas de escritura y lectura de las distintas funciones que se encargan de encender y seleccionar los colores del LED, para cada caso.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS				
106335	+229	0x0B	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x72650F	-38	OK			
106336	+29772	0x13	0x50655B5C	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Write_Command		CRC	RSSI (dBm)	FCS
106337	+318	0x13	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x726E7A	-36	OK			
106338	+29683	0x1B	0x50655B5C	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Read_Req		CRC	RSSI (dBm)	FCS
106339	+285	0x1B	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x72650F	-37	OK			

Figura 117. Tramas de encendido del LED

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS				
112305	+229	0x23	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x726E7A	-43	OK			
112306	+29772	0x06	0x50655B5C	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Write_Command		CRC	RSSI (dBm)	FCS
112307	+318	0x06	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x72650F	-36	OK			
112308	+29683	0x0E	0x50655B5C	M->S	OK	L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	ATT_Read_Req		CRC	RSSI (dBm)	FCS
112309	+285	0x0E	0x50655B5C	S->M	OK	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x726E7A	-37	OK			

Figura 118. Tramas de cambio de color del LED a rojo

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS						
P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	LLID	NESH	SN	MD	PDU-Length	L2CAP-Header	ATT_Write_Command			CRC	RSSI (dBm)	FCS	
115972	+230 =298368717	0x09	0x5065585C	S->M	OK	Empty PDU	1	1	0	0	0	0x726E7A	-38	OK					
115973	+29770 =2983717487	0x11	0x5065585C	M->S	OK	L2CAP-S	2	1	1	0	11	0x0007	0x0004	0x52	0x0019	00 00 FF 00	0x266C92	-31	OK
115974	+317 =2983717804	0x11	0x5065585C	S->M	OK	Empty PDU	1	0	1	0	0	0x72650F	-38	OK					
115975	+29687 =2983747491	0x19	0x5065585C	M->S	OK	L2CAP-S	2	0	0	0	7	0x0003	0x0004	0x0A	0x0019		0x90DBA6	-31	OK
115976	+285 =2983747776	0x19	0x5065585C	S->M	OK	Empty PDU	1	1	0	0	0	0x726E7A	-37	OK					

Figura 119. Tramas de cambio de color del LED a verde

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				CRC	RSSI (dBm)	FCS						
P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	LLID	NESH	SN	MD	PDU-Length	L2CAP-Header	ATT_Write_Command			CRC	RSSI (dBm)	FCS	
9258	+229 =27799698	0x23	0x5065726D	S->M	OK	Empty PDU	1	0	1	0	0	0xF7FCD4	-30	OK					
9259	+29773 =278029471	0x0D	0x5065726D	M->S	OK	L2CAP-S	2	0	0	0	11	0x0007	0x0004	0x52	0x0019	00 00 00 FF	0x6E8BCA	-33	OK
9260	+318 =278029789	0x0D	0x5065726D	S->M	OK	Empty PDU	1	1	0	0	0	0xF7F7A1	-30	OK					
9261	+29684 =278059473	0x1C	0x5065726D	M->S	OK	L2CAP-S	2	1	1	0	7	0x0003	0x0004	0x0A	0x0019		0xA8A7CB	-30	OK
9262	+286 =278059759	0x1C	0x5065726D	S->M	OK	Empty PDU	1	0	1	0	0	0xF7FCD4	-30	OK					

Figura 120. Tramas de cambio de color del LED a azul

Todas las tramas de solicitud de lectura obtienen la correspondiente trama de respuesta de lectura, tal y como se observa en la Figura 121.

P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	Data Header				L2CAP Header	ATT_Read_Req		CRC	RSSI (dBm)	FCS			
P.nbr.	Time (us)	Channel	Access Address	Direction	ACK Status	Data Type	LLID	NESH	SN	MD	PDU-Length	L2CAP-Header	ChanId	Opcode	AttHandle	AttValue	CRC	RSSI (dBm)	FCS
9261	+29684 =278059473	0x1C	0x5065726D	M->S	OK	L2CAP-S	2	1	1	0	7	0x0003	0x0004	0x0A	0x0019		0xA8A7CB	-30	OK
9262	+286 =278059759	0x1C	0x5065726D	S->M	OK	Empty PDU	1	0	1	0	0	0xF7FCD4	-30	OK					
9263	+29716 =278089475	0x06	0x5065726D	M->S	OK	Empty PDU	1	0	0	0	0	0xF7F172	-33	OK					
9264	+230 =278089705	0x06	0x5065726D	S->M	OK	L2CAP-S	2	1	0	0	9	0x0005	0x0004	0x0B	00 00 00 FF		0xD2FB36	-30	OK
9265	+29770 =278119475	0x15	0x5065726D	M->S	OK	Empty PDU	1	1	1	0	0	0xF7FA07	-30	OK					

Figura 121. Tramas de solicitud y respuesta de lectura de una característica

Esta trama de respuesta de lectura es enviada por parte del dispositivo BLE *Peripheral* a la petición de lectura que ha solicitado el dispositivo BLE *Central*. La Figura 121 muestra la respuesta de lectura de la trama de selección de color del LED a azul, donde el *master* ha solicitado leer el atributo de la característica con el identificador 0x0019, y el *slave* ha devuelto el valor del atributo

solicitado. Este hecho se puede confirmar al comparar el valor del atributo de la Figura 120 con el valor del atributo de la Figura 121 y ver que coinciden, siendo este valor en ambos casos 00 00 00 FF.

De todas las imágenes presentadas, se puede concluir que las tramas de escritura desde el dispositivo BLE *Central* escriben en el atributo de la característica con el identificador 0x019, que es el correspondiente a la característica que controla los estados del LED, y que a su vez coincide con el identificador de la característica que utiliza la trama de lectura para solicitar el dato al dispositivo BLE *Peripheral*. Lo que confirma que los procesos de lectura y escritura son correctos.

Para comprobar el funcionamiento del código por parte de Alexa se va a recurrir al uso de los *test* ya comentados en capítulos anteriores. En este caso, se van a realizar cinco *test*, uno por cada estado que pueda adoptar el LED. Los estados que puede adoptar el LED son los siguientes: encendido, apagado y color rojo, verde o azul.

Los *test* que se van a realizar son muy parecidos a los explicados anteriormente, cuyas diferencias radican en el identificador de la *skill* a la que se está llamando, por el simple hecho de ser *skills* diferentes, la función a la que se llama, y los *slots* que se utilizan, para distinguir una acción de otra, tal y como se muestra en la Figura 122.

```
1 ▾ {
2 ▾  "session": {🔴},
13  "version": "1.0",
14 ▾  "request": {
15    "locale": "en-US",
16    "timestamp": "2016-10-27T21:06:28Z",
17    "type": "IntentRequest",
18    "requestId": "amzn1.echo-api.request.78d5c8c9-6ad2-40a5-bcac-d53b62b6fe8e",
19 ▾    "intent": {
20 ▾      "slots": {
21 ▾        "color": {
22 ▾          "name": "color",
23 ▾          "value": "blue"
24 ▾        }
25 ▾      },
26    "name": "setLightColor"
27  }
28 },
29 ▾ "context": {🔴}
47 }
```

Figura 122. Test de la función *setLightColor*

La Figura 123 muestra el código utilizado para testear la función *setLightState*. Para el resto de los casos se utilizan estos dos *test*, pero se cambian los valores de los *slots* utilizados por los otros posibles valores con el fin de validar experimentalmente todos los casos.


```
1 {
2   "session": {  
13   "version": "1.0",  
14   "request": {  
15     "locale": "en-US",  
16     "timestamp": "2016-10-27T21:06:28Z",  
17     "type": "IntentRequest",  
18     "requestId": "amzn1.echo-api.request.78d5c8c9-6ad2-40a5-bcac-d53b62b6fe8e",  
19     "intent": {  
20       "slots": {  
21         "state": {  
22           "name": "state",  
23           "value": "on"  
24         }  
25       },  
26       "name": "setLightState"  
27     }  
28   },  
29   "context": {  
47 }  
}
```

Figura 123. Test de la función setLightState

La Figura 124, la Figura 125, la Figura 126, la Figura 127 y la Figura 128 muestran el resultado para los diferentes test de cada función, dependiendo del valor del *slot* utilizado

DuoCandle Limitación Cualificadores Acciones stateIntent Probar Guardar

Resultado de la ejecución: correcta (Registros)

▼ Detalles

En el área siguiente se muestra el resultado devuelto por la ejecución de la función. [Obtenga más información](#) sobre cómo devolver resultados de la función.

```
{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "PlainText",
      "text": "The candle is now down"
    },
    "shouldEndSession": true,
    "card": {
      "type": "Simple",
```

Figura 124. Resultado del test de la función setLightState con el slot down

DuoCandle Limitación Cualificadores ▼ Acciones ▼ stateIntent ▼ Probar Guardar

✔ Resultado de la ejecución: correcta (Registros) ✕

▼ Detalles

En el área siguiente se muestra el resultado devuelto por la ejecución de la función. [Obtenga más información](#) sobre cómo devolver resultados de la función.

```

{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "PlainText",
      "text": "The candle is now on"
    },
    "shouldEndSession": true,
    "card": {
      "type": "Simple",

```

Resumen

Código SHA-256	ID de solicitud
3v83kFyOzfbYDuxVTnccnqEiKYM36xPOwhCNQOEfhAo=	a4435642-ff21-11e8-b75b-d7a8eaf54e6e
Duración	Duración facturada
10694.95 ms	10700 ms

Figura 125..Resultado del test de la función setLightState con el slot on

DuoCandle Limitación Cualificadores ▼ Acciones ▼ colorIntent ▼ Probar Guardar

✔ Resultado de la ejecución: correcta (Registros) ✕

▼ Detalles

En el área siguiente se muestra el resultado devuelto por la ejecución de la función. [Obtenga más información](#) sobre cómo devolver resultados de la función.

```

{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "PlainText",
      "text": "The candle is now red"
    },
    "shouldEndSession": true,
    "card": {
      "type": "Simple",

```

Resumen

Código SHA-256	ID de solicitud
3v83kFyOzfbYDuxVTnccnqEiKYM36xPOwhCNQOEfhAo=	0c0c175a-ff22-11e8-a3ae-73487a3ecce4
Duración	Duración facturada
362.06 ms	400 ms

Figura 126. Resultado del test de la función setLightColor con el slot red

DuoCandle Limitación Cualificadores ▼ Acciones ▼ colorIntent ▼ Probar Guardar

✔ Resultado de la ejecución: correcta (Registros) ✕

▼ Detalles

En el área siguiente se muestra el resultado devuelto por la ejecución de la función. [Obtenga más información](#) sobre cómo devolver resultados de la función.

```

},
"shouldEndSession": true,
"card": {
  "type": "Simple",
  "title": "duo candle",
  "content": "The candle is now green"
},
"sessionAttributes": {}
}

```

Resumen

Código SHA-256	ID de solicitud
3v83kFyOzfbYDuxVTnccnqEiKYM36xPOwhCNQOEfhAo=	249e86c9-ff22-11e8-81be-73dabafcc51f
Duración	Duración facturada
10334.44 ms	10400 ms

Figura 127. Resultado del test de la función setLightColor con el slot green

DuoCandle Limitación Cualificadores ▼ Acciones ▼ colorIntent ▼ Probar Guardar

✓ Resultado de la ejecución: correcta (Registros) ✕

▼ Detalles

En el área siguiente se muestra el resultado devuelto por la ejecución de la función. [Obtenga más información](#) sobre cómo devolver resultados de la función.

```

{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "PlainText",
      "text": "The candle is now blue"
    },
    "shouldEndSession": true,
    "card": {
      "type": "Simple",

```

Resumen

Código SHA-256	ID de solicitud
3v83kFyOzfbyDuxVTnccnqEiKYM36xPOwhCNQOEfhAo=	e5306d87-ff21-11e8-b39c-9d076f4b46c3
Duración	Duración facturada
13357.92 ms	13400 ms

Figura 128. Resultado del test de la función `setLightColor` con el slot `blue`

Para todos los casos el resultado es el esperado, ya que en todos y cada uno de ellos la función responde con el texto que contiene la respuesta que emitiría el dispositivo *Echo Dot* de Alexa en su discurso. Para poder concluir que las *skills* de Alexa, y por tanto la solución final, funcionan correctamente, es necesario validar su funcionamiento por medio de comandos de voz. Para ello, se hará uso del dispositivo *Echo Dot*. Los comandos de voz que se van a utilizar para comprobar el correcto funcionamiento de la solución son los siguientes:

- *Alexa, open duo candle*: comando para iniciar sesión.
- *Alexa, ask duo candle to turn [on | off]*: comando para encender o apagar el LED.
- *Alexa, ask duo candle to turn [red | green | blue]*: comando para seleccionar el color del LED.

Tras utilizar estos comandos de voz se comprueba que la solución final funciona correctamente, cumpliendo con las condiciones preestablecidas. Por tanto, se puede afirmar que se han alcanzado los objetivos establecidos al comienzo del presente TFG, y que el usuario es capaz de alterar el estado actual del LED del dispositivo *PLAYBULB Candle* mediante la voz.

9. Conclusiones

El análisis de los resultados obtenidos tras el proceso de validación funcional concluye que se han cumplido los objetivos principales de este Trabajo Fin de Grado, utilizando una estrategia enfocada en la realización de la pasarela HW/SW mediante la separación de las tareas reconocidas.

En primer lugar, se realizó un estudio de las características propias del dispositivo *Photon* que se utilizó durante la primera parte del TFG para llevar a cabo un desarrollo inicial, seguidamente, se llevó a cabo un estudio acerca del funcionamiento de los servicios de Alexa y su interacción con el usuario, el cual aportó los conocimientos necesarios para poder elaborar una *skill* de Alexa adecuadamente.

Después del análisis del contexto, se ha diseñado una pasarela inicial utilizando el dispositivo *Photon*, que carece de conectividad BLE, pero proporciona un punto de partida para la creación del *firmware* que se utilizará en la plataforma HW/SW final. Esta pasarela inicial sentó las bases de conocimiento de la plataforma de interconexión de dispositivos y desgranó el funcionamiento de los servicios que ofrecen las *skills* de Alexa.

Los conocimientos adquiridos tras la elaboración de la plataforma HW/SW inicial, representaron el marco ideal para proceder a añadir la conectividad BLE de la que dispone el dispositivo *RedBear DUO*. Para ello, se realizó un estudio de las características propias del dispositivo *DUO*, el cual fue mucho más fácil de comprender dadas sus similitudes con el dispositivo *Photon* y a la adquisición de experiencia por parte del estudiante.

Se ha realizado un análisis de la teoría relacionada con la tecnología BLE para caracterizar de manera correcta el código incluido en el *firmware* del dispositivo *RedBear DUO*, y así permitir que un usuario, mediante su voz, pueda interactuar con un dispositivo BLE.

Desde una visión genérica, el estudio realizado en el desarrollo de este Trabajo Fin de Grado ha llevado consigo un aprendizaje ligado a la importancia de utilizar un asistente virtual configurable, como es Alexa, ya que sin la amplia variedad de configuraciones que le aportan las *skills* el desarrollo de este proyecto no sería posible.

También es remarcable la diferenciación del código que existe entre la primera implementación de la plataforma HW/SW mediante el dispositivo *Photon*, y el código utilizado en la última versión del *firmware* del dispositivo *RedBear DUO*, ya que en la última versión se observa una manera más eficiente de procesar los datos, que corresponde a la adquisición de experiencia por parte del estudiante.

El *firmware* desarrollado para la elaboración del presente TFG se ha realizado de forma que esté capacitado para interactuar con un tipo en concreto de dispositivo BLE, siendo la plataforma HW/SW desarrollada incapaz adaptarse y operar en consecuencia, si se cambia el dispositivo BLE utilizado por otro que no presente las mismas características. A pesar de este hecho, a lo largo del desarrollo del presente Trabajo Fin de Grado, se han descrito y utilizado las herramientas necesarias para adaptar el proyecto a cualquier dispositivo BLE.

El estudio de la tecnología BLE, junto con la implementación de ésta en dispositivos IoT, ha creado una idea fundada de las capacidades disponibles en el campo de los asistentes virtuales para posibles integraciones con esta tecnología inalámbrica.

Por último, este TFG representa una forma más de las posibilidades que representan los pequeños dispositivos inteligentes que, utilizados en combinación con diferentes tecnologías inalámbricas, permiten crear sistemas baratos, eficientes y con una escalabilidad muy alta.

Referencias

- [1] Internet of things – ITU [Online] Available: https://www.itu.int/dms_pub/itu-s/opb/pol/S-POL-IR.IT-2005-SUM-PDF-E.pdf
- [2] ATZORI, Luigi; IERA, Antonio; et al. The Internet of Things: A Survey, 2010
- [3] ASHTON, Kevin. That ‘Internet of Things’ thing. RFID Journal, 2009
- [4] MIORANDI, Daniele; et al. Internet of Things: Vision, Applications and Research Challenges. Ad Hoc Networks, 2012
- [5] Internet of Things – Cisco [Online]. Available: www.cisco.com/c/dam/global/es.../Internet_of_Things_IoT_IBSG_0411FINAL.pdf [Last Access: Enero 2018].
- [6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. ALEDhari and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys & Tutorials 17(4), pp. 2347-2376. 2015. DOI: 10.1109/COMST.2015.2444095.
- [7] Fog computing. [Online]. Available: https://en.wikipedia.org/wiki/Fog_computing. [Last Acces: Enero 2018]
- [8] Carlos, C; Dharma P; AD HOC SENSOR NETWORKS, Theory and applications , Feb 2011
- [9] H.-J. Yoo et al., ”Body area network: Technology, solutions, and standardisation”, in IEEE ISSCC Dig. Tech Papers, Feb. 2011, p. 531
- [10] Bluetooth SIG, Specification of the Bluetooth System v4.0, June 30th, 2010[Online]. Available: www.Bluetooth.org. [Last Acces: Enero 2018].
- [11] Brent A. Miller. Chatschik Bisdikian. Bluetooth Revealed 2nd. Prentice Hall PTR Upper Saddle River, NJ, USA ©2001
- [12] Wi-Fi (802.11b) and Bluetooth: enabling coexistence. Lansford, J. Sep/Oct 2001. 15 , Issue: 5.
- [13] Bluetooth v4.0: la futura solución inalámbrica de bajo consumo, Ricard Morales Pedro, 8 de Julio de 2011.
- [14] Who (or What) Is Alexa? We Explain Amazon’s Digital Assistant. [Online]. Available: <http://www.tomsguide.com/us/amazon-alexa-faq,review-4016.html>. [Last Access: Enero 2018].

- [15] Facebook empieza a probar un asistente personal llamado `M` en tu Messenger – 20minutos.es [Online]. Available: <http://www.20minutos.es/noticia/2542970/0/facebook/asistente-personal-pruebas/messenger/> [Last Access: Enero 2018].
- [16] iOS – Siri – Apple (ES). [Online] Available: <http://www.apple.com/es/ios/siri/> [Last Access: Enero 2018]
- [17] Google Now. [Online] Available: <https://www.google.com/intl/es/landing/now/> [Last Access: Enero 2018].
- [18] Cortana – pret-a-porter, novia y fiesta. Shop Online. [Online] Available:; <https://www.cortana.es/> [Last Access: Enero 2018].
- [19] Alexa: Keyword Research, Competitor Analysis, & Website Ranking. [Online] Available: <http://www.alexa.com/>. [Last Access: Enero 2018].
- [20] Managing Device Discovery for Your Alexa Smart Home Skill.[Online]. Available: <https://developer.amazon.com/blogs/post/Tx480XF5QCA6IN/managing-device-discovery-for-your-alexa-smart-home-skill>. [Last Access: Enero 2018].
- [21] Particle Photon Datasheet. [Online]. Available: [https://docs.particle.io/datasheets/photon-\(wifi\)/photon-datasheet/#functional-description](https://docs.particle.io/datasheets/photon-(wifi)/photon-datasheet/#functional-description) [Last Access: Junio 2018]
- [22] Particle Photon Datasheet [Online]. Available: [https://docs.particle.io/datasheets/photon-\(wifi\)/photon-datasheet/#pin-and-button-definition](https://docs.particle.io/datasheets/photon-(wifi)/photon-datasheet/#pin-and-button-definition) [Last Access: Junio 2018]
- [23] Particle Docs [Online]. Available: <https://docs.particle.io/guide/getting-started/start/photon/#connect-your-photon> [Last Access: Junio 2018]
- [24] Particle Docs: Devices Modes [Online]. Available: <https://docs.particle.io/guide/getting-started/modes/photon/> [Last Access: Septiembre 2018].
- [25] Particle Docs: Web IDE [Online]. Available: <https://docs.particle.io/tutorials/developer-tools/build/core/> [Last Access Diciembre 2018]
- [26] Particle Docs: Particle Cloud [Online]. Available: <https://docs.particle.io/reference/device-cloud/api/> [Last Access: Diciembre 2018]
- [27] Developing Alexa Skills: The Big Nerd Ranch Guide [Online] Available: <https://github.com/bignerdranch/developing-alexa-skills-solutions/blob/master/coursebook/coursebook.pdf> [Last Access: Diciembre 2018]

- [28] Robin Heydon. *Bluetooth Low Energy. The Developer's Handbook*. Crawfordsville, Indiana. Prentice Hall, 2012.
- [29] PLAYBULB Candle. [Online] Available: <https://www.mipow.com/products/playbulb-candle> [Last Access: Diciembre 2018]
- [30] RedBear Lab. [Online]. Available: www.redbearlab.com. [Last Access: Junio 2018].
- [31] RedBear Introduction [Online]. Available: https://github.com/redbear/Duo/blob/master/docs/duo_introduction.md [Last Access: Junio 2018].
- [32] RedBear Out of Box Experience [Online]. Available: https://github.com/redbear/Duo/blob/master/docs/out_of_box_experience.md [Last Access: Septiembre 2018].
- [33] DUO: Windows Driver Installation Guide [Online]. Available: https://github.com/redbear/Duo/blob/master/docs/windows_driver_installation_guide.md [Last Access: Septiembre 2018].
- [34] Smart Protocol Packet Sniffer [Online]. Available: <http://www.ti.com/tool/PACKET-SNIFFER> [Last Access: Diciembre 2018]
- [35] CC2540 USB Dongle. [Online]. Available: <http://www.ti.com/tool/TIDC-CC2540-BLE-USB#> [Last Access: Diciembre 2018]

Pliego de Condiciones

Se procede a presentar las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. Se realizará una diferenciación entre el conjunto de herramientas *hardware*, *software* y *firmware* empleadas para llevar a cabo su realización.

Condiciones *Hardware*

En la Tabla 3 se indican el conjunto de dispositivos *hardware* utilizados, con su modelo correspondiente

Dispositivo/herramienta	Modelo	Fabricante/comerciante
Ordenador portátil	Asus F556U Series	Asus
Photon	Photon con pines soldados	Particle
RedBear DUO	RedBear DUO con pines soldados	RedBear
Dispositivo BLE	PLAYBULB Candle	Mipow
Amazon Echo Dot	Echo Dot	Amazon
Teléfono móvil	Iphone 6	Apple
Packet Sniffer	CC2540 USB Dongle	Texas Instruments

Tabla 3. Condiciones *Hardware*

Condiciones *Software*

En la Tabla 4 se recoge las aplicaciones software utilizadas, con su versión correspondiente

Software	Versión	Desarrollador
Sistema operativo portátil	Microsoft Windows 7 Home Premium 64-bit Edition	Microsoft
Sistema operativo teléfono móvil	iOS11	Apple
Microsoft Office	2016	Microsoft
Microsoft Visio	2016	Microsoft
Microsoft Project	2016	Microsoft
Particle IDE		Particle
Google Chrome	V67.0.3396.99/ 64 bit	Google
Adobe Reader	V11.0.21.18	Adobe Systems Software Ireland Ltd.
nRF Connect	V4.19.2	Nordic Semiconductor
PLAYBULBX	v1.6.7	miPOW Ltd

Tabla 4. Condiciones *Software*

Condiciones *firmware*

En la Tabla 5 se indica el *firmware* utilizado para cada dispositivo, y su versión correspondiente:

Firmware	Versión
Photon	v0.6.2
RedBear Duo	v0.3.1
PLAYBULB Candle	V2.3.0.31

Tabla 5. Condiciones Firmware

Presupuesto

En este capítulo se abordará el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. El presupuesto, asimismo, está compuesto por las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividida a su vez en:
 - Amortización del material *hardware*.
 - Amortización del material *software*.
- Redacción de la documentación.

Una vez analizados los puntos que componen el presupuesto se procederá a aplicar los impuestos vigentes y se procederá a la obtención del coste total del Trabajo Fin de Grado.

Trabajo tarifado por tiempo empleado

En esta sección se contabilizarán los gastos que corresponden a la mano de obra según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Con este objetivo se ha utilizado la fórmula de la ecuación 1.

$$H = Ct \cdot 74,88 \cdot Hn + Ct \cdot 96,72 \cdot He \quad (1)$$

Donde:

- H: Honorarios totales por el tiempo dedicado.
- *Hn*: Número de horas normales trabajadas dentro de la jornada laboral.
- *Ct*: Factor de corrección que depende del número de horas trabajadas.
- *He*: Número de horas especiales trabajadas.

Se han invertido un total de 300 horas en la realización del presente Trabajo Fin de Grado. Casi todas ellas se han realizado dentro del horario normal, por lo que el número de horas especiales es 60. De acuerdo con lo establecido por el COITT (Colegio Oficial de Ingenieros de Telecomunicación), el factor de corrección *Ct* que se aplica para 300 horas trabajadas es de 0,60, como se aprecia en la Tabla 6.

Horas	Factor de corrección
Hasta 36	1,00
Exceso de 36 hasta 72	0,90
Exceso de 72 hasta 108	0,80
Exceso de 108 hasta	0,70

144	
Exceso de 144 hasta 180	0,65
Exceso de 180 hasta 360	0,60

Tabla 6. Factor de corrección

Por tanto, haciendo uso de la ecuación 1:

$$- H = 0,6 \cdot 74,88 \cdot 240 + 0,6 \cdot 96,72 \cdot 60 = 14.264,64\text{€}$$

El trabajo tarifado por tiempo empleado asciende a la cantidad de catorce mil doscientos sesenta y cuatro euros con sesenta y cuatro céntimos.

Amortización del inmovilizado material

El inmovilizado material se compone de los recursos hardware y software empleados para la realización de este Trabajo Fin de Grado.

Se ha utilizado un sistema de amortización lineal, en el que se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula como se indica en la ecuación 2.

$$\mathbf{Cuota\ anual} = \frac{\mathbf{Valor\ de\ adquisición} - \mathbf{Valor\ residual}}{\mathbf{Número\ de\ años\ de\ vida\ útil}} \quad (2)$$

El Valor residual corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil. A continuación, se analizará la amortización de los recursos hardware y software utilizados.

Amortización del material hardware

En la Tabla 7 se especifican los elementos hardware amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Dispositivo/Herramienta	Valor de adquisición	Valor de amortización
Ordenador Asus F556U	880,00 €	115,00 €
Teléfono móvil Iphone 6	600,00 €	103,25 €
<i>Packet Sniffer</i>	40,00 €	40,00 €
<i>Photon</i>	20,00€	20,00€
<i>RedBear DUO</i>	19,27 €	19,27 €
<i>PLAYBULB Candle</i>	24,68€	24,68€

<i>Echo Dot</i>	30,00€	30,00€
Total	1613,95 €	352,20 €

Tabla 7. Amortización del material hardware

Debido al bajo precio de los dispositivos, el conjunto formado por los dispositivos IoT, y el *packet sniffer*, así como los dispositivos *Amazon Echo Dot* y *PLAYBULB Candle*, su valor de amortización coincide con su valor de adquisición. Para el resto de los elementos hardware se ha utilizado la fórmula indicada por la ecuación 2.

El coste total del material hardware asciende a trescientos cincuenta y dos euros con veinte céntimos.

Amortización del material software

En la Tabla 8 se indican los elementos software amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Software	Valor de adquisición	Valor de amortización
Sistema operativo Windows 7 Home Premium	Licencia UPLGC	0,00 €
Particle IDE	0.00 € / Software acceso libre	0,00 €
Microsoft Visio	Licencia UPLGC	0,00 €
Microsoft Project	Licencia UPLGC	0,00 €
Microsoft Office	Licencia UPLGC	0,00 €
Google Chrome	0.00 € / Software acceso libre	0,00 €
Adobe Reader	0.00 € / Software acceso libre	0,00 €
Total	0,00 €	0,00 €

Tabla 8. Amortización del material software

El coste total asociado al material software es de cero euros.

Sumando los costes del inmovilizado del material hardware y software se obtiene el coste total de inmovilizado material, indicado en la Tabla 9.

Concepto	Coste
Material hardware	352,20 €
Material software	0,00 €
Total	352,20 €

Tabla 9. Coste total inmovilizado material

Redacción del trabajo

Para el cálculo del coste asociado a la redacción de la memoria del presente trabajo se ha usado la fórmula mostrada en la ecuación 3.

$$R = 0,07 \cdot P \cdot Cn \quad (3)$$

Donde:

- R: Honorarios por la redacción del trabajo.
- P: Presupuesto.
- **Cn**: Coeficiente de ponderación en función del presupuesto.

El cálculo del presupuesto se realiza mediante la suma de los costes del trabajo tarifado por tiempo empleado y la amortización del inmovilizado material. En la Tabla 10 se indica este cálculo.

Concepto	Coste
Trabajo tarifado por tiempo empleado	14.264,64€
Amortización del inmovilizado material	352,20 €
Total	14.616,84 €

Tabla 10. Presupuesto

A continuación, se analiza el coeficiente de ponderación *Cn* para este presupuesto. Según el COITT, para sueldos inferiores a 30.050,00 € les corresponde un valor de 1,00. En consecuencia, el coste asociado a la redacción del Trabajo Fin de Grado se indica en la ecuación 4:

$$R = 0,07 \cdot 14.616,84 \cdot 1 = 1023,18 \text{ €} \quad (4)$$

Aplicación de impuestos y coste total

La realización del presente Trabajo Fin de Grado está gravada con el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7%). En la Tabla 11 se indica el cálculo del presupuesto total del Trabajo Fin de Grado aplicando el impuesto.

Concepto	Coste
Trabajo tarifado por tiempo empleado	14.264,64 €
Amortización del inmovilizado material	352,20 €
Redacción del trabajo	1023,18 €
Total (sin IGIC)	15.640,02 €
IGIC (7%)	1094,80 €
Total	16.734,82 €

Tabla 11. Presupuesto total del Trabajo Fin de Grado

Por tanto, el presupuesto total del Trabajo Fin de Grado “Integración de dispositivos BLE con el Asistente Virtual Alexa mediante tecnología IoT” asciende a dieciséis mil setecientos treinta y cuatro euros con ochenta y dos céntimos.

En Las Palmas de Gran Canaria, a 15 de enero de 2019.

Fdo. Javier Araña Melián

Anexo Contenido de los ficheros adjuntos

Adjunto a la memoria del presente Trabajo Fin de Grado se encuentra disponible un Compact Disc Read-Only Memory (CD-ROM). En este anexo se especifica el contenido incluido:

En el directorio Memoria Trabajo Fin de Grado se encuentra la memoria del TFG “Desarrollo de una plataforma para la interacción con la interfaz OBD-II utilizando BLE” en lengua española y en formato PDF.

En el directorio Solución Inicial se encuentra el fichero *Light.ino* que contiene el código del dispositivo *Photon* y un directorio denominado *Alexa*, el cual a su vez presenta dos nuevos directorios en su interior. El primero de ellos, denominado *speechAssets*, contiene el archivo *IntentSchema.json*, que contiene la estructura utilizada para desarrollar las *skills* de la solución inicial, el archivo *LIST_OF_SELECTIONS.txt*, que contiene los estados del *slot select*, el archivo *LIST_OF_STATES.txt*, que contiene los estados del *slot state*, y el archivo *SampleUtterances.txt*, que contiene los *utterances* que pueden utilizarse para referirse a los *intent* de la *skill*. El segundo directorio que aparece en la carpeta *Alexa*, se denomina *src*, contiene el archivo *AlexaSkill.js*, que contiene las funcionalidades básicas para el desarrollo de la *skill*, el archivo *index.js*, que contiene la implementación de la *skill service*, el archivo *package.json*, que contiene la información referente a los paquetes de *Node.js* necesarios para el correcto funcionamiento de la *skill*, y la carpeta *node_modules* que contiene los archivos referente a los paquetes de *Node.js* que se han especificado anteriormente en el archivo *package.json*.

En el directorio Solución Final se encuentra el fichero *blecentralDUO.ino* que contiene el código del dispositivo *DUO* y un directorio denominado *Alexa*, el cual a su vez presenta dos nuevos directorios en su interior. El primero de ellos, denominado *speechAssets*, contiene el archivo *IntentSchema.json*, que contiene la estructura utilizada para desarrollar las *skills* de la solución final, el archivo *LIST_OF_COLORS.txt*, que contiene los estados del *slot color*, el archivo *LIST_OF_STATES.txt*, que contiene los estados del *slot state*, y el archivo *SampleUtterances.txt*, que contiene los *utterances* que pueden utilizarse para referirse a los *intent* de la *skill*. El segundo directorio que aparece en la carpeta *Alexa*, se denomina *src*, contiene el archivo *AlexaSkill.js*, que contiene las funcionalidades básicas para el desarrollo de la *skill*, el archivo *index.js*, que contiene la implementación de la *skill service*, el archivo *package.json*, que contiene la información referente a los paquetes de *Node.js* necesarios para el correcto funcionamiento de la *skill*, y la carpeta *node_modules* que contiene los archivos referente a los paquetes de *Node.js* que se han especificado anteriormente en el archivo *package.json*.